Skeletal Semantics and their Interpretations

Martin Bodin, Philippa Gardner, Thomas Jensen, Alan Schmitt

Skeletal Semantics and their Interpretations

MARTIN BODIN, Imperial College
PHILIPPA GARDNER, Imperial College
THOMAS JENSEN, Inria
ALAN SCHMITT, Inria

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.

Authors' addresses: Martin Bodin, Imperial College; Philippa Gardner, Imperial College; Thomas Jensen, Inria; Alan Schmitt, Inria.

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

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

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

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

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

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

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 manually. Over the years, the challenge has been to generate such an abstract semantics from the concrete semantics and prove consistency. There has been much work on providing recipes for such an endeavour. For example, Van Horn and Might [Van Horn and Might 2010] provide a recipe for constructing abstract semantics from well-known abstract machines. Midtgaard and Jensen [Midtgaard and Jensen 2008] describe a systematic derivation of control flow analyses.

The initial work on abstract interpretation [Cousot and Cousot 1977] assumed that the concrete semantics was given by a transition system. Schmidt [Schmidt 1995, 1997a] instead proposed a rule-based approach, assuming that the concrete semantics was given as a big-step operational semantics. He lifted the Galois connection from concrete and abstract domains to concrete and abstract derivations, and proved consistency. This approach produced an abstract semantics close to the concrete semantics. It was easier to build, but it was still not systematic. Recently, Bodin *et al.* [Bodin et al. 2015] applied Schmidt's approach to a restricted setting called *pretty-big-step* semantics [Charguéraud 2013]. They identified a general pretty-big-step rule format, demonstrated that it was possible to generate abstract rules from concrete rules in a systematic way, and provided a general consistency result. In contrast, we have introduced a meta-language for skeletal semantics, that can be instantiated to any language with a big-step operational semantics, with several general interpretations consistency results.

We demonstrate our skeletal semantics in action using the simple conditional statement from the While language. Consider the usual concrete rules associated with the conditional statement in Figure 1, and the abstract rules in Figure 2, supposing that the booleans are abstracted by the usual four-valued lattice given by $\{\mathit{true}^\#, \mathit{false}^\#, \top_{bool}, \bot_{bool}\}$. These abstract rules are intuitively correct, but they are first built in an ad hoc way and then shown to be related to the concrete rules using a Galois connection. More generally, the systematic construction of abstract rules from concrete rules requires a deep understanding of how the analysed programming language evaluates expressions: in a case like a vanilla While language, this is quite straightforward; but for a complex language such as JavaScript [Bodin et al. 2014; ECMA 2018; Maffeis et al. 2008], the relationship between the concrete and abstract semantics can be difficult to get right.

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

Fig. 3. Skeleton for the *if* construct

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

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

We define the skeletal semantics, providing a general meta-theory for defining language semantics, in Section 2. Figure 3 shows the skeleton associated with the *if* construct, with generic 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 identifies the required subcomputations associated with x_{t_1}, x_{t_2}, and x_{t_3}. The skeleton stitches these H constructs together, using the internal symbolic variable x_{f_1} and the branching which identifies paths through the skeleton using the filters `isTrue` and `isFalse`. Such a skeleton thus explicitly describes both the data flow and the control flow associated with a language construct, identifying the common pattern underlying the concrete and abstract rules.

We provide a general definition of interpretation for our skeletal semantics in Section 3 and study four generic interpretations:

- A simple well-formedness interpretation which states that the stitching of the skeleton in Figure 3 respects the sorting of the basic constructs.
- The concrete interpretation (Section 4) which intuitively picks one path from each branching of the skeleton, corresponding to the two rules of Figure 1.
- 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.
- 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.

We also define notions of *consistency* between interpretations (Section 3.2). The shared structure of our different interpretations greatly eases the proof process. We use our notions of consistency to show that the abstract interpretation is correct in relation to the concrete interpretation, and that any solution to the constraints generated must be a correct abstract semantics.

We instantiate our skeletal semantics to a WHILE language throughout the paper, and demonstrate how classic proof techniques based on an abstract interpretation of WHILE can be captured with our approach (Section 6). We however emphasise that skeletons and interpretations, as well as their consistency proofs, are generic and can be applied to any programming language. Most proofs have been moved to the anonymous supplementary material for space reasons.

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

2 SKELETAL SEMANTICS

2.1 Terms

The first ingredient of a skeletal semantics is the syntactic terms and their sorts. We assume given a countable set of *sorts*, ranged over by s, separated into *base sorts* and *program sorts*. We also assume given a countable set of term variables, ranged over by x_t, and a finite set of constructors, ranged 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 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 $\textsc{Name}(c(x_{t_1}..x_{t_n})) \coloneqq S$, where Name is the skeleton name, t is a term, and S is the *skeleton body*.

$$
\begin{aligned}
\textsc{LitInt}(\mathit{const}\,(x_t)) &\coloneqq [\texttt{litToVal}\,(x_t)\ ?\triangleright\ x_o] \\
\textsc{Var}(\mathit{var}\,(x_t)) &\coloneqq [\texttt{read}\,(x_t, x_\sigma)\ ?\triangleright\ x_o] \\
\textsc{Add}(x_{t_1} + x_{t_2}) &\coloneqq \left[\begin{array}{l} H\,(x_\sigma, x_{t_1}, x_{f_1})\,;\texttt{isInt}\,(x_{f_1})\ ?\triangleright\ x_{f_{1'}}; H\,(x_\sigma, x_{t_2}, x_{f_2})\,;\texttt{isInt}\,(x_{f_2})\ ?\triangleright\ x_{f_{2'}}; \\ \quad \texttt{add}\,(x_{f_{1'}}, x_{f_{2'}})\ ?\triangleright\ x_o \end{array}\right] \\
\textsc{Eq}(x_{t_1} = x_{t_2}) &\coloneqq \left[\begin{array}{l} H\,(x_\sigma, x_{t_1}, x_{f_1})\,;\texttt{isInt}\,(x_{f_1})\ ?\triangleright\ x_{f_{1'}}; H\,(x_\sigma, x_{t_2}, x_{f_2})\,;\texttt{isInt}\,(x_{f_2})\ ?\triangleright\ x_{f_{2'}}; \\ \quad \texttt{eq}\,(x_{f_{1'}}, x_{f_{2'}})\ ?\triangleright\ x_o \end{array}\right] \\
\textsc{Neg}(\neg x_t) &\coloneqq \left[H\,(x_\sigma, x_t, x_{f_1})\,;\texttt{isBool}\,(x_{f_1})\ ?\triangleright\ x_{f_2}; \texttt{neg}\,(x_{f_2})\ ?\triangleright\ x_o\right] \\
\textsc{Skip}(\mathit{skip}) &\coloneqq [\texttt{id}\,(x_\sigma)\ ?\triangleright\ x_o] \\
\textsc{Asn}(x_{t_1} \coloneqq x_{t_2}) &\coloneqq \left[H\,(x_\sigma, x_{t_2}, x_{f_1})\,;\texttt{write}\,(x_{t_1}, x_\sigma, x_{f_1})\ ?\triangleright\ x_o\right] \\
\textsc{Seq}(x_{t_1};\ x_{t_2}) &\coloneqq \left[H\,(x_\sigma, x_{t_1}, x_{f_1})\,; H\,(x_{f_1}, x_{t_2}, x_o)\right] \\
\textsc{If}(\mathit{if}\ x_{t_1}\ x_{t_2}\ x_{t_3}) &\coloneqq \left[H\,(x_\sigma, x_{t_1}, x_{f_1})\,;\texttt{isBool}\,(x_{f_1})\ ?\triangleright\ x_{f_{1'}}; \begin{pmatrix} \texttt{isTrue}\,(x_{f_{1'}})\,; H\,(x_\sigma, x_{t_2}, x_o) \\ \texttt{isFalse}\,(x_{f_{1'}})\,; H\,(x_\sigma, x_{t_3}, x_o) \end{pmatrix}_{\{x_o\}}\right] \\
\textsc{While}(\mathit{while}\ x_{t_1}\ x_{t_2}) &\coloneqq \left[\begin{array}{l} H\,(x_\sigma, x_{t_1}, x_{f_1})\,;\texttt{isBool}\,(x_{f_1})\ ?\triangleright\ x_{f_{1'}}; \\ \quad \begin{pmatrix} \texttt{isTrue}\,(x_{f_{1'}})\,; H\,(x_\sigma, x_{t_2}, x_{f_2})\,; H\,(x_{f_2}, \mathit{while}\ x_{t_1}\ x_{t_2}, x_o) \\ \texttt{isFalse}\,(x_{f_{1'}})\,;\texttt{id}\,(x_\sigma)\ ?\triangleright\ x_o \end{pmatrix}_{\{x_o\}} \end{array}\right]
\end{aligned}
$$

Fig. 6. Skeletal semantics for While

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

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 x_{f_2}; or a *filter* $F(x_1..x_n)\ ?\triangleright\ (y_1..y_m)$ 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 $\mathit{in}(s)$ as its input flow sort and $\mathit{out}(s)$ as its output flow sort. Their definitions are for our running example are given in Figure 7.

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

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

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;
- $\left[\!\left[\bigoplus_n\right]\!\right]^I_V\ (O, \Sigma) \Downarrow \Sigma'$ 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.

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

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

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

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

Fig. 9. WF Interpretation

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 $\{x_\sigma \mapsto in(s) + x_{t_1} \mapsto Sort(t_1)..x_{t_n} \mapsto Sort(t_n)\}$, and with $\mathcal{D} = dom(\Gamma)$.

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.

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

Definition 3.2. Interpretations I_1 and I_2 are *existentially consistent* if for any S, Σ_1, Σ_2, and O_1, such 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 $OKout(O_1, O_2)$.

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, if $OKst(\Sigma_1, \Sigma_2)$, $[\![L]\!]^{I_1}\,(\Sigma_1) \Downarrow O_1$ and $[\![L]\!]^{I_2}\,(\Sigma_2) \Downarrow O_2$, then $OKout(O_1, O_2)$.

3.3 Proving Consistency

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

Lemma 3.4. *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) $\left[\!\left[H(x_{f_1}, t, x_{f_2})\right]\!\right]^{I_1}\,(\Sigma_1) \Downarrow \Sigma_1'$ *implies there is an Σ_2' such that* $\left[\!\left[H(x_{f_1}, t, x_{f_2})\right]\!\right]^{I_2}\,(\Sigma_2) \Downarrow \Sigma_2'$ *and* $OKst(\Sigma_1', \Sigma_2')$;
(3) $[\![F(x_1..x_n)\ ?\triangleright\ (y_1..y_m)]\!]^{I_1}\,(\Sigma_1) \Downarrow \Sigma_1'$, *implies there is an Σ_2' such that* $[\![F(x_1..x_n)\ ?\triangleright\ (y_1..y_m)]\!]^{I_2}\,(\Sigma_2) \Downarrow \Sigma_2'$ *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 $\forall i \in dom(O_1).OKout(O_1(i), O_2(i))$,* $\left[\!\left[\bigoplus_n\right]\!\right]_V^I\,(O_1, \Sigma_1) \Downarrow \Sigma_1'$ *implies there is an Σ_2' such that* $\left[\!\left[\bigoplus_n\right]\!\right]_V^I\,(O_2, \Sigma_2) \Downarrow \Sigma_2'$ *and* $OKst(\Sigma_1', \Sigma_2')$;

then I_1 and I_2 are existentially consistent.

Lemma 3.5. *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) $\left[\!\left[H(x_{f_1}, t, x_{f_2})\right]\!\right]^{I_1}\,(\Sigma_1) \Downarrow \Sigma_1'$ *and* $\left[\!\left[H(x_{f_1}, t, x_{f_2})\right]\!\right]^{I_2}\,(\Sigma_2) \Downarrow \Sigma_2'$ *implies* $OKst(\Sigma_1', \Sigma_2')$;
(3) $[\![F(x_1..x_n)\ ?\triangleright\ (y_1..y_m)]\!]^{I_1}\,(\Sigma_1) \Downarrow \Sigma_1'$ *and* $[\![F(x_1..x_n)\ ?\triangleright\ (y_1..y_m)]\!]^{I_2}\,(\Sigma_2) \Downarrow \Sigma_2'$, *implies* $OKst(\Sigma_1', \Sigma_2')$;
(4) *for any O_1, O_2 of domain a subset of $[1..n]$ and such that $\forall i \in dom(O_1) \cap dom(O_2).OKout(O_1(i), O_2(i))$,* $\left[\!\left[\bigoplus_n\right]\!\right]_V^I\,(O_1, \Sigma_1) \Downarrow \Sigma_1'$ *and* $\left[\!\left[\bigoplus_n\right]\!\right]_V^I\,(O_2, \Sigma_2) \Downarrow \Sigma_2'$ *implies* $OKst(\Sigma_1', \Sigma_2')$;

then I_1 and I_2 are universally consistent.

4 CONCRETE INTERPRETATION

We now define an interpretation used to compute a big-step evaluation semantics in the form of a *triple set*: a set of triples (also called *judgements*) of the form *(state, term, result)*. For each base sort we assume given a set of *base terms*, and for each flow sort a set of *values*. We write $t : s$ to state that base term t has base sort s, and $v : s$ to state that value v has flow sort s.

For each filter $F(x_1..x_n)\ ?\triangleright\ (y_1..y_m)$ such that $fsort(F) = (s_1..s_n) \to (s_1'..s_m')$, we assume given an interpretation $[\![F]\!]$ which is a relation between elements of $(s_1..s_n)$ and elements of $(s_1'..s_m')$. We write $[\![F]\!]\,(v_1..v_n) \Downarrow (v_1'..v_m')$ to state it relates $(v_1..v_n)$ to $(v_1'..v_m')$.

The input state of a concrete interpretation is a pair comprising

- an environment Σ mapping term variables to closed terms and flow variables to values,
- a set T of triples of value, closed term, and value, representing already known judgements and used to give meaning to the sub-derivations $H(x_{f_1}, t, x_{f_2})$.

The interpretation result maps term variables to closed terms and flow variables to values.

We define the concrete interpretation in Figure 10. For the empty skeleton body, it simply returns its environment. For a hook $H(x_{f_1}, t, x_{f_2})$, it looks up in the triple set a known computation for $\Sigma(x_{f_1})$ and $\Sigma(t)$ whose result is v, and it continues binding x_{f_2} to v. Note that if the language is non-deterministic, there may be several such values and one is picked. For a filter F, one uses its

$$\Longrightarrow \llbracket [] \rrbracket (\Sigma, T) \Downarrow \Sigma$$

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

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

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

Fig. 10. Concrete Interpretation

interpretation with the input $(\Sigma(x_1)..\Sigma(x_n))$. As filter interpretations are relations, there may be several results as well. Finally, to merge branches, the interpretation picks a branch that successfully returned a result and extends its environment accordingly.

Running Example. We instantiate the base sort *ident* with strings and *lit* with integers. We instantiate the flow sort *int* with integers, *bool* with booleans, *val* with the disjoint union *int* + *bool*, and *state* with a partial function from strings to *val*. The concrete interpretation of the filters are the following partial functions: `litToVal`(i): injects i from *int* to *int* + *bool*, `read`(id, st): applies st to id (since st is a partial function, it may not return a result), `isInt`(v): matches v in the disjoint union *int* + *bool*, returns v if it is in *int*, `add`(i_1, i_2): returns the integer addition of i_1 and 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 *int* + *bool* otherwise, `isBool`(v): matches v in the disjoint union *int* + *bool*, returns v if it is in *bool*, `write`(id, st, v): returns the partial function mapping id to v and any other id' to $st(id')$, `id`(st): returns st, `isTrue`(b): returns () if b = `true`, `isFalse`(b): returns () if b = `false`. In the rest of the paper, we directly write x for *var*(x) and n for *const*(n) in the examples.

4.1 Consistency of WF and Concrete Interpretations

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 $v : out(s)$.

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 $\Sigma(x) : \Gamma(x)$.

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

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

4.2 Concrete Derivations

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.

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

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

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. □

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

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.

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

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

Proof. The set of triple sets ordered by inclusion is a CPO, and $\mathcal{H}$ is continuous on this CPO. We conclude by Kleene fixpoint theorem. □

Lemma 4.7. *The concrete semantics $\Downarrow$ is well-formed.*

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

5 ABSTRACT INTERPRETATION

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

5.1 Abstract Domains

An abstract interpretation of a set of skeletons must define abstract domains for all the terms and flow sorts used in the skeleton bodies, ending with abstract semantic states and abstract results.

Elements in the abstract domains represent sets of values in the corresponding concrete domain (they are related through the concretion function γ introduced below). The abstract interpretation framework is designed to be parametric in the choice of abstract domains for base values such as integers, booleans, and program states. All we require is that each abstract domain for sort s is a partial order $\sqsubseteq$ with a least element, denoted $\bot_s$, representing the empty set. For example, the lattice of intervals can be used as an abstract domains for integers, with $\bot_{int}$ being the empty interval. Similarly, a state that maps program variables to integer values can be abstracted as a mapping from variables to intervals, or as a polyhedron that defines linear relations between program variables.

Skeletal variables can also range over terms. For each program or base sort s, we define an abstract domain by imposing a flat partial order on the set of terms of that sort (*i.e.*, we relate a term to itself and no other term) and by adding a $\bot_s$ element, smaller than all terms of that sort. Abstract

base terms include every concrete base term, they may also include additional terms that denote sets of concrete base terms. To lighten notations, we sometimes omit the sort in $\bot$ in an equality. In this case, $v^\# = \bot$ should be read $v^\# = \bot_{Sort(v^\#)}$ and, $v^\# \neq \bot$ should be read $v^\# \neq \bot_{Sort(v^\#)}$.

5.2 Abstract Interpretation of Skeletons

In addition to the abstract domains, an abstract interpretation must specify its input and output states, and how the empty skeleton body, hooks, filters, and the merging of branches are interpreted. For each filter symbol F of signature $(s_1..s_n) \to (s'_1..s'_m)$ we assume give a total function $[\![F]\!]^\#$ from the domain corresponding to $(s_1..s_n)$ to the domain corresponding to $(s'_1..s'_m)$. A filter interpretation may return $\bot$ to state it is not defined for that input.

The input state of an abstract interpretation is a triple $(f, \Sigma^\#, T^\#)$ comprising a *flag* f, an abstract environment $\Sigma^\#$ (a mapping from skeletal variables to abstract terms and values), and a set of abstract semantic triples $T^\#$ that gives a semantics to hooks. A flag is either $\bot$ or $\top$ and it indicates whether it has been determined the current skeleton does not apply ($\bot$) or that it may still apply ($\top$). The output state of an abstract interpretation is a flag and an abstract semantic where skeletal variables hold the result of the abstract interpretation. Figure 11 defines the abstract semantics.

The abstract interpretation of an empty list of hypotheses just returns the flag and environment from its input. There are three cases for the interpretation of a hook $H(x_{f_1}, t, x_{f_2})$. If we have determined that the skeleton does not apply, we just set x_{f_2} to $\bot$ of the correct sort. In the two other cases, we need to have a triple $(\sigma^\#, t^\#, v^\#)$ from $T^\#$ such that $\Sigma^\#(x_{f_2}) \sqsubseteq \sigma^\#$ and $\Sigma^\#(t) \sqsubseteq t^\#$. 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.

The abstract interpretation of a filter $F(x_1..x_n)\ ?\triangleright\ (y_1..y_m)$ also has three cases. If we know the skeleton does not apply, we just bind the output variables $(y_1..y_m)$ to the appropriate $\bot$ depending on the signature of F. Otherwise, we apply the filter interpretation to an approximation of the arguments as given by the environment. If the result is $\bot$, we know the skeleton does not apply and switch the flag to $\bot$, as well as extend the environment with $\bot$ of the correct sort. Otherwise, we keep the flag as $\top$ and extend the environment to an approximation of the result of the filter.

For the merging operator, we interpret every possible branch and collect their results in $\mathcal{O}$. If all branching have the $\bot$ flag (either because the $\bot$ flag was set before their interpretation, which would then be propagated, or because they newly returned it), then the skeleton does not apply and we set the flag accordingly, extending the environment with mappings from the shared skeletal variables V to $\bot$ of the correct sort. Otherwise, we collect all branches that have a $\top$ flag. They must all return abstract environments that agree on the shared variables (which is why the approximations in the filter and hook cases are useful, to ensure this is possible), and we extend the current environment with this common environment.

The key difference between the abstract and concrete interpretations is how the different results are merged in case of branching. The concrete semantics picks one of them, whereas the abstract semantics requires all branches that provided a result to agree. This is because the goal of the abstract semantics is to infer abstract semantic triples that are valid statements about all possible resulting states, i.e., about all possible concrete choices in case of branching.

5.3 Consistency of WF and Abstract Interpretations

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

$$\implies [\![[]]\!]^{\#}\left(f, \Sigma^{\#}, T^{\#}\right) \Downarrow \left(f, \Sigma^{\#}\right)$$

$$\Sigma^{\#\prime} = \Sigma^{\#} + x_{f_2} \mapsto \bot_{out(Sort(\Sigma^{\#}(t)))} \implies \left[\!\left[H\left(x_{f_1}, t, x_{f_2}\right)\right]\!\right]^{\#}\left(\bot, \Sigma^{\#}, T^{\#}\right) \Downarrow \left(\bot, \Sigma^{\#\prime}, T^{\#}\right)$$

$$\begin{pmatrix} \Sigma^{\#}\left(x_{f_1}\right) \sqsubseteq \sigma^{\#} \\ \Sigma^{\#}\left(t\right) \sqsubseteq t^{\#} \\ \left(\sigma^{\#}, t^{\#}, \bot\right) \in T^{\#} \\ \Sigma^{\#\prime} = \Sigma^{\#} + x_{f_2} \mapsto \bot_{out(Sort(\Sigma^{\#}(t)))} \end{pmatrix} \implies \left[\!\left[H\left(x_{f_1}, t, x_{f_2}\right)\right]\!\right]^{\#}\left(\top, \Sigma^{\#}, T^{\#}\right) \Downarrow \left(\bot, \Sigma^{\#\prime}, T^{\#}\right)$$

$$\begin{pmatrix} \Sigma^{\#}\left(x_{f_1}\right) \sqsubseteq \sigma^{\#} \\ \Sigma^{\#}\left(t\right) \sqsubseteq t^{\#} \\ \left(\sigma^{\#}, t^{\#}, v^{\#}\right) \in T^{\#} \\ v^{\#} \sqsubseteq v^{\#\prime} \\ \Sigma^{\#\prime} = \Sigma^{\#} + x_{f_2} \mapsto v^{\#\prime} \end{pmatrix} \implies \left[\!\left[H\left(x_{f_1}, t, x_{f_2}\right)\right]\!\right]^{\#}\left(\top, \Sigma^{\#}, T^{\#}\right) \Downarrow \left(\top, \Sigma^{\#\prime}, T^{\#}\right)$$

$$\begin{pmatrix} fsort\left(F\right) = \left(s_1^{\prime}..s_n^{\prime}\right) \rightarrow \left(s_1..s_m\right) \\ \Sigma^{\#\prime} = \Sigma^{\#} + y_1 \mapsto \bot_{s_1}..y_m \mapsto \bot_{s_m} \end{pmatrix} \implies [\![F\left(x_1..x_n\right) ?\triangleright \left(y_1..y_m\right)]\!]^{\#}\left(\bot, \Sigma^{\#}, T^{\#}\right) \Downarrow \left(\bot, \Sigma^{\#\prime}, T^{\#}\right)$$

$$\begin{pmatrix} \left(\Sigma^{\#}\left(x_1\right)..\Sigma^{\#}\left(x_n\right)\right) \sqsubseteq \left(v_1^{\#}..v_n^{\#}\right) \\ [\![F]\!]^{\#}\left(v_1^{\#}..v_n^{\#}\right) = \bot \\ fsort\left(F\right) = \left(s_1^{\prime}..s_n^{\prime}\right) \rightarrow \left(s_1..s_m\right) \\ \Sigma^{\#\prime} = \Sigma^{\#} + y_1 \mapsto \bot_{s_1}..y_m \mapsto \bot_{s_m} \end{pmatrix} \implies [\![F\left(x_1..x_n\right) ?\triangleright \left(y_1..y_m\right)]\!]^{\#}\left(\top, \Sigma^{\#}, T^{\#}\right) \Downarrow \left(\bot, \Sigma^{\#\prime}, T^{\#}\right)$$

$$\begin{pmatrix} \left(\Sigma^{\#}\left(x_1\right)..\Sigma^{\#}\left(x_n\right)\right) \sqsubseteq \left(v_1^{\#}..v_n^{\#}\right) \\ [\![F]\!]^{\#}\left(v_1^{\#}..v_n^{\#}\right) \sqsubseteq \left(v_1^{\#\prime}..v_m^{\#\prime}\right) \\ \Sigma^{\#\prime} = \Sigma^{\#} + y_1 \mapsto v_1^{\#\prime}..y_m \mapsto v_m^{\#\prime} \end{pmatrix} \implies [\![F\left(x_1..x_n\right) ?\triangleright \left(y_1..y_m\right)]\!]^{\#}\left(\top, \Sigma^{\#}, T^{\#}\right) \Downarrow \left(\top, \Sigma^{\#\prime}, T^{\#}\right)$$

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

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

Fig. 11. Abstract Interpretation

We define $OKout(\Gamma, (f, \Sigma^{\#}))$ as follows: $dom(\Gamma) = dom(\Sigma^{\#})$ and for any $x \in dom(\Gamma)$ we have $\Sigma^{\#}(x) : \Gamma(x)$.

Lemma 5.1. *The well-formedness and abstract interpretations are universally consistent.*

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

Fig. 12. The abstract immediate consequence operator

5.4 Abstract Derivations

We define the abstract immediate consequence operator from well-formed triple sets to triple sets in Figure 12. As in the concrete case, the immediate consequence describes how to assemble skeletons.

Lemma 5.2. *The functional $\mathcal{H}^{\#}$ is monotonic.*

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. □

Lemma 5.3. *If $T^{\#}$ is a well-formed triple set, then $\mathcal{H}^{\#}(T^{\#})$ is a well-formed triple set.*

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. □

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

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 $v \in \gamma(v^\#)$.

We define $OKst(\Sigma, T)(f, \Sigma^\#, T^\#)$ as follows: $f = \top$, $dom(\Sigma) = dom(\Sigma^\#)$, for any $x \in dom(\Sigma)$, we have $\Sigma(x) \in \gamma(\Sigma^\#(x))$, and T and $T^\#$ are well formed and consistent.

We define $OKout\Sigma(f, \Sigma^\#)$ as follows: $f = \top$, $dom(\Sigma) = dom(\Sigma^\#)$, and for any $x \in dom(\Sigma)$, we have $\Sigma(x) \in \gamma(\Sigma^\#(x))$.

Lemma 5.7. *The concrete and abstract interpretations are universally consistent.*

Lemma 5.8. *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.*

We finally show that the abstract semantics is correct in relation to the concrete semantics. In a nutshell, for any triple in the concrete semantics $\Downarrow$ (the smallest fixpoint of $\mathcal{H}$) and any triple in the abstract semantics $\Downarrow^\#$ (the largest fixpoint of $\mathcal{H}^\#$), if the input states and terms are related, then the output values are related. Formally, we have the following.

Definition 5.9. An abstract triple set $T^\#$ is *correct* if it is well-formed and consistent with $\Downarrow$.

Theorem 5.10. *$\Downarrow^\#$ is correct.*

Proof. We prove by induction on k that $\mathcal{H}^k(\emptyset)$ and $\Downarrow^\#$ are well formed and consistent. The check that $\Downarrow^\#$ is well formed is simply Lemma 5.4.

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 $\Downarrow^\#$ are well formed and consistent. By Lemma 5.8 we have $\mathcal{H}^{n+1}(\emptyset)$ and $\mathcal{H}^\#(\Downarrow^\#) = \Downarrow^\#$ are well formed and consistent, as required.

To conclude, we apply Lemma 4.6. □

Note that in the previous theorem we only use the fact that $\Downarrow^\#$ is a fixpoint: it does not have to be the greatest fixpoint.

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

Fig. 13. Abstract interpretation of filters

6 DERIVING PROOF TECHNIQUES FROM AN ABSTRACT SEMANTICS

We present in this section several proof techniques derived from an abstract semantics and instantiated in our While language.

6.1 Example: Interval analysis of While

To give a concrete example of an abstract interpretation we design a value analysis of the language While in the style of Schmidt's Abstract Interpretation of Natural Semantics [Schmidt 1995]. We have the following flow sorts in the semantic definition (cf. Figure 7): *int*, *bool*, *val*, and *state*.

We describe an analysis in which integers are approximated by intervals, ordered by inclusion. Writing $[n,m]$ for the interval of integers between n and m (with the convention that $[n,m] = \emptyset$ if $m < n$), we can define the abstract domains for each of the flow sort as follows:

$$
\begin{aligned}
int^\# &= ([n,m] : n \in \mathbf{Z} \cup \{-\infty\} \wedge m \in \mathbf{Z} \cup \{+\infty\}) & val^\# &= int^\# \times bool^\# \\
bool^\# &= \{\bot_{bool}, true^\#, false^\#, \top_{bool}\} & state^\# &= ident^\# \to val^\#
\end{aligned}
$$

We abstract identifiers by themselves, thus $ident^\# = ident$, with only the trivial (reflexive) ordering. The abstract domain of Booleans is (isomorphic to) the set of subsets of Booleans, ordered by inclusion. The abstract domain of values is the defined as the Cartesian product, ordered component-wise, of the abstract domain of integers and Booleans, where each component gives an approximation of the concrete value, *provided* that the value is of the corresponding sort. States are represented

as mappings from identifiers to values, ordered pointwise. Identifiers that have not been defined 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.

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

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

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 $c(t_1..t_n)$, we take the corresponding skeleton $\textsc{Name}(c(x_{t_1}..x_{t_n})) := S$ in the semantics and apply the general abstract interpretation to the skeleton body S. This results in a series of conditions for a triple to be valid that will form the hypotheses of the inference rules.

Rule for addition. As a first example, we derive a rule for analysing arithmetic expressions such 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,

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

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

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

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

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

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

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

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

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

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:

$$\begin{array}{lll} & \left(\sigma^\#, \mathit{if}\ t_1\ t_2\ t_3, \sigma_o^\#\right) \in \mathcal{H}^\#\left(T^\#\right) & \\ \Leftrightarrow & \left[\!\left[H\left(x_\sigma, x_{t_1}, x_{f_1}\right); \texttt{isBool}\left(x_{f_1}\right) \mathbin{?\triangleright} x_{f_{1'}}; \begin{pmatrix} \texttt{isTrue}\left(x_{f_{1'}}\right); H\left(x_\sigma, x_{t_2}, x_o\right) \\ \texttt{isFalse}\left(x_{f_{1'}}\right); H\left(x_\sigma, x_{t_3}, x_o\right) \end{pmatrix}_{\{x_o\}} \right]\!\right]^\# \left(\top, \Sigma_1^\#, T^\#\right) \Downarrow \left(f, \Sigma_o^\#\right) & \\ & \quad \wedge \quad \Sigma_1^\# = x_\sigma \mapsto \sigma^\# + x_{t_1} \mapsto t_1 + x_{t_2} \mapsto t_2 + x_{t_3} \mapsto t_3 \quad \wedge \quad \sigma_o^\# = \Sigma_o^\#\left(x_o\right) & \\ \Leftarrow & \left(\sigma^\#, t_1, v_1^\#\right) \in T^\# \quad \wedge \quad v_1^\# = \left(i_1^\#, b_1^\#\right) \quad \wedge & \\ & \left[\!\left[\begin{pmatrix} \texttt{isTrue}\left(x_{f_{1'}}\right); H\left(x_\sigma, x_{t_2}, x_o\right) \\ \texttt{isFalse}\left(x_{f_{1'}}\right); H\left(x_\sigma, x_{t_3}, x_o\right) \end{pmatrix}_{\{x_o\}} \right]\!\right]^\# \left(\top, \Sigma_2^\#, T^\#\right) \Downarrow \left(f, \Sigma_o^\#\right) & \\ & \quad \wedge \quad \Sigma_1^\# = x_\sigma \mapsto \sigma^\# + x_{t_1} \mapsto t_1 + x_{t_2} \mapsto t_2 + x_{t_3} \mapsto t_3 + x_{f_1} \mapsto v_1^\# + x_{f_{1'}} \mapsto b_1^\# & \end{array}$$

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:

$$\begin{array}{lll} & \left(\sigma^\#, \mathit{if}\ t_1\ t_2\ t_3, \sigma_o^\#\right) \in \mathcal{H}^\#\left(T^\#\right) & \\ \Leftarrow & \left(\sigma^\#, t_1, v_1^\#\right) \in T^\# \quad \wedge \quad \left(\sigma^\#, t_2, \sigma_2^\#\right) \in T^\# \quad \wedge \quad \left(\sigma^\#, t_3, \sigma_3^\#\right) \in T^\# & \\ & \quad \wedge \quad v_1^\# = \left(i_1^\#, \top_{bool}\right) \quad \wedge \quad \sigma_2^\# \sqsubseteq \sigma_o^\# \quad \wedge \quad \sigma_3^\# \sqsubseteq \sigma_o^\# & \end{array}$$

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 $\Sigma^\#(x_{f_{1'}}) = \mathit{true}^\#$.

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

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.

$$\begin{array}{rl} & \left(\sigma^\#, \mathit{if}\ t_1\ t_2\ t_3, \sigma_o^\#\right) \in \mathcal{H}^\#\left(T^\#\right) \\ \Leftarrow & \left(\sigma^\#, t_1, v_1^\#\right) \in T^\# \quad \wedge \quad \left(\sigma^\#, t_2, \sigma_o^\#\right) \in T^\# \quad \wedge \quad v_1^\# = \left(i_1^\#, \mathit{true}^\#\right) \end{array}$$

We can rewrite as above this implication as a rule.

$$\frac{\sigma^\# \vdash t_1 : \left(i_1^\#, \mathit{true}^\#\right) \qquad \sigma^\# \vdash t_2 : \sigma_2^\#}{\sigma^\# \vdash \mathit{if}\ t_1\ t_2\ t_3 : \sigma_2^\#}$$

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

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

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

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

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

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

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

This implication means that given any $T_0^\#$ such that $T_0^\# \subseteq \mathcal{H}^\#(T_0^\#)$, that associates t_1 in the state $\sigma^\#$ with a result (that is that there exists $v^\#$ such that $(\sigma^\#, t_1, v^\#) \in T_0^\#$), and such that $(\sigma^\#, t_2, \sigma^\#) \in T_0^\#$, 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 $T_0^\# \subseteq \mathcal{H}^\#(T_1^\#)$, 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 triple in $T_1^\#$ is correct in relation to $\Downarrow$. In other words, the following familiar rule is admissible.

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

6.3 State Splitting

As another example of the use of the abstract interpretation, we show how to extend the abstract semantics to obtain more precise results. Our motivating example is t: *while* $\neg(\mathsf{x} = 0)\ \mathsf{x} := \mathsf{x} - 1$ for which we want to show that the triple $(\mathsf{x} \mapsto [0, \infty], t, \mathsf{x} \mapsto 0)$ is correct (we simplify notation and 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, observe that in the rule for While, the same state is used to run the expression and the statement, hence the return value of the expression is not reflected in the state (it may only prevent a branch from being taken). Communicating information from an expression back to a state is a non-trivial problem which depends on the language considered, but we can help the abstract interpretation by splitting the state in three parts: $\{(\mathsf{x} \mapsto 0, t, \mathsf{x} \mapsto 0), (\mathsf{x} \mapsto [1, \infty], t, \mathsf{x} \mapsto 0), (\mathsf{x} \mapsto [0, \infty], t, \mathsf{x} \mapsto 0)\}$. Let $T^\#$ be the set of triples (listed below) obtained from adding triples for every sub-expression 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 $(\mathsf{x} \mapsto [0, \infty], t, \mathsf{x} \mapsto 0)$ to evaluate the recursive while term). However there is still one of the three triples that cannot be derived, *viz.*, $(\mathsf{x} \mapsto [0, \infty], t, \mathsf{x} \mapsto 0) \in \mathcal{H}^\#(T^\#)$.

To derive this third triple, we introduce a proof technique called *state splitting* to obtain a more precise abstract semantic. The core idea of the technique is that if the state $\sigma^\#$ of a triple $(\sigma^\#, t^\#, v^\#)$ is covered by the states of some triples $(\sigma_1^\#, t^\#, v^\#)..(\sigma_n^\#, t^\#, v^\#)$, in the sense that $\gamma(\sigma^\#) \subseteq \gamma(\sigma_1^\#) \cup .. \cup \gamma(\sigma_n^\#)$, then we may use $(\sigma^\#, t^\#, v^\#)$ in the input triple set $T^\#$ of $\mathcal{H}^\#(T^\#)$ *without* having to show that $(\sigma^\#, t^\#, v^\#)$ is in the resulting triple set.

Formally, we first define a function *Sp* from triple sets to triple sets that add such triples.

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 $\cdot$ and neutral element ϵ. We define a subterm operator $t@\mathsf{pp}$ as follows.

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

We assume given a function *Gent* that for a given term t_0 states the set of program points for which constraints will be generated. It typically consists of the set of executable subterms of t_0. We require the program points of $Gent(t_0)$ to be executable: if $\mathsf{pp} \in Gent(t_0)$, then $t_0@\mathsf{pp} = c(t_1..t_n)$. Requirement 2.2 enforces the existence of a skeleton for this term.

We next define a partial operator PP_{t_0} that associates program points to the terms occurring in the hooks of a skeleton. Formally, if $PP_{t_0}(\mathsf{pp}, \mathrm{N}, t) = \mathsf{pp}'$, then (1) skeleton N is applicable: $\mathsf{pp} \in Gent(t_0)$, $t_0@\mathsf{pp} = c(t_1..t_n)$, and N is of the form $\mathrm{N}(c(x_{t_1}..x_{t_n})) \coloneqq S$, (2) the hook $H(x_{f_1}, t, x_{f_2})$ occurs in S, and (3) the resulting program point is part of the set of explored program points: $\mathsf{pp}' \in Gent(t_0)$ and $t_0@\mathsf{pp}' = (x_{t_1} \mapsto t_1..x_{t_n} \mapsto t_n)(t)$.

Constraints are either of the form $[x = x']$, $[x \sqsubseteq x']$, or $[x : s]$, where x and x' are variables and s a sort. We generate variable names in constraints of the form pp-x. The constraint generation function *Gen* that takes a program t_0 and returns the set of constraints generated by t_0 is defined as

$$Gen\,(t_0) \triangleq \bigcup C \cup \left\{ \begin{array}{c} [\mathsf{pp}\text{-}x_\sigma : in\,(Sort\,(t_0@\mathsf{pp}))]\,, \\ [\mathsf{pp}\text{-}x_o : out\,(Sort\,(t_0@\mathsf{pp}))]\,, \\ \forall i \in [1..n].\ [\mathsf{pp}\text{–}x_{t_i} = t_i] \end{array} \right\} \quad \left| \begin{array}{c} \mathsf{pp} \in Gent\,(t_0) \wedge t_0@\mathsf{pp} = c\,(t_1..t_n) \\ \mathrm{N}\big(c\,(x_{t_1}..x_{t_n})\big) \coloneqq S \in Rules \\ [\![S]\!]^c\,(\mathrm{N}, \mathsf{pp}, \emptyset) \Downarrow C \\ \mathcal{D}_\mathrm{N}\,(\emptyset) = \{x_{t_1}..x_{t_n}, x_\sigma\} \wedge x_o \in \mathcal{D}_\mathrm{N}\,(C) \end{array} \right.$$

For each skeleton N we define a function $\mathcal{D}_\mathrm{N}$ that maps sets of constraints to sets of skeletal variables. This is not necessary for the constraint generation but is used to prove consistency between constraints and the abstract semantics.

The constraint generation interpretation of skeletons $[\![S]\!]^c$ is given in Figure 14. The rule for hooks generates constraints for connecting the input state pp'-x_σ with the flow variable holding the input state in the hook pp–x_{f_1}, and the resulting output state of the hook with the output of the hook. Each filter comes with a constraint generation function $[\![F]\!]^c$ specific to the analysis of that filter. We require that the constraints generated for that filter agree with the abstract 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

[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.

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

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

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

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

Fig. 14. Constraint Generation

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 generate constraints for each branch and return the union of these constraint sets.

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.

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:*

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

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

Discussion. The constraints we generate are *path-insensitive*: they do not capture the fact that when a filter does not hold, the rest of the skeleton does not matter. Constraints can be path-sensitive by letting the state of the interpretation be a pair consisting of a set *Stop* of constraint sets representing pathways in the skeleton that are stopped, similar to the $\bot$ flag in the abstract interpretation, and one constraint set *Run* representing all the running paths. When a filter is encountered, the *Run* set is added to *Stop* with the additional constraint that the filter returns $\bot$. The usual constraints for the filter are added to *Run*. In a nutshell, we duplicate constraints for each filter: once when it does not hold, and once when it may hold. At the end of the interpretation, the

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 $t_0 = \mathit{while}\ \neg(\mathtt{x} = 0)\ \mathtt{x} := \mathtt{x} - 1$. 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

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

The constraint generation will produce the constraints

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

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

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

The constraints on $1\text{-}x_\sigma$ and $1\text{-}x_o$ are generated when considering the program point 1, corresponding to the evaluation of $\neg(\mathtt{x} = 0)$ (corresponding to the skeleton Neg). As stated, the set of all generated constraints is large; it is provided in the supplementary material.

8 RELATED WORK

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

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

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

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

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

The $\mathbb{K}$ framework has goals similar to ours: derive verifiers from operational semantics, correct by construction. A key difference is that the semantics of the $\mathbb{K}$ specification tool is complex and 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 skeletal approach to semantics. With skeletal semantics we can capture many language constructs, including higher-order and object-oriented features. We have so far not studied how distributed and interactive computation fit into the framework, nor have we studied how the formalism scales to bigger languages. For instance, the specification of JavaScript [ECMA 2018] is written in a style where the whole behaviour of each construct of the language is a single definition, making it straightforward to express as a skeletal semantics.

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

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

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

REFERENCES

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 the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2014)*. San Diego, CA, USA, 87–100.

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.

Arthur Charguéraud. 2013. Pretty-big-step Semantics. In *European Symposium on Programming*. Springer, 41–60.

Martin Churchill, Peter D Mosses, Neil Sculthorpe, and Paolo Torrini. 2015. Reusable Components of Semantic Specifications. In *Transactions on Aspect-Oriented Software Development XII*. Springer, 132–179.

Patrick Cousot. 1999. The Calculational Design of a Generic Abstract Interpreter. In *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 238–252.

Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. 2016. Semantics-Based Program Verifiers for All Languages. In *Proc. of the 31th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*. ACM, 74–91.

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. 2013. Views: Compositional Reasoning for Concurrent Programs. In *Proc. of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. ACM, 287–300.

ECMA. 2018. ECMAScript 2018 Language Specification (ECMA-262, 9th edition). (June 2018). https://www.ecma-international.org/ecma-262/9.0/index.html

Philippa Gardner, Sergio Maffeis, and Gareth Smith. 2012. Towards a Program Logic for JavaScript. *ACM SIGPLAN Notices* 47, 1 (2012), 31–44.

Robert Harper, Furio Honsell, and Gordon D. Plotkin. 1987. A Framework for Defining Logics. In *Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987*. IEEE Computer Society, 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.