Working with Operational Semantics as a Programming Language

By Sebastian Hyberts
and Kári Frederiksen

Supervised by Hans Hüttel
Spring 2022

Institute for Computer Science
Aalborg University

Abstract: Treating the specifications of Structural Operational Semantics as a programming language opens up for a variety of analysis methods which are commonly used in research surrounding programming languages. This paper gives a type system that uses unification to ensure specifications are well-typed and uses graph-based analyses to ensure the bindings and premises within rules are logically ordered. This opens up for research into the role of having a compiler-like tool available when learning and working with specifications of operational semantics.

Contents

1 Introduction

Formal specifications of languages are typically complex systems and understanding the semantics of a language by reading the rules usually involves understanding a lot of definitions and their interplay. Any typos and small errors in these semantic definition might slow down unfamiliar readers and can present an especially unnecessary challenge when initially reading and building the mental model of the definitions. However mistakes do occur and Klein et al. (2012) found that in nine published papers examined, every single paper contained some errors that all were found during the process of mechanising and examining the semantics.

1.1 Errors in Structural Operational Semantics

To understand how the problem of errors in formal semantics can be tackled, we must first gain an understanding of some of the errors present. We will restrict the focus to Structural Operational Semantics (SOS) in both the big- and small-step styles. SOS is commonly used for semantics and is characterised by being syntax-directed and based on defining transition systems and a finite set of inference rules, which together form a semantic specification(Plotkin 2004). We can divide errors in operational semantics into two categories, namely local and global errors.

- Local errors relate to a single rule, both with and without the involvement of the specification-wide definitions.
- Global errors are errors involving multiple rules.

1.1.1 Incorrect Configurations

When working with operational semantics, all transition rules are part of defined transition systems. When students of operational semantics initially start learning about these systems, we have found it common to see rules that do not follow the definition of transition systems.

$$\textsc{Seq}: \quad \frac{\langle S_1, \mathit{env}, \mathit{sto}\rangle \to \langle S_1', \mathit{env}', \mathit{sto}'\rangle \quad \mathit{evn}' = \mathit{evn}[a \mapsto 1]}{\langle S_1 \mathbin{;} S_2, \mathit{env}\rangle \to S_1' \mathbin{;} S_2}$$

Figure 1: Rule with wrong variable-use and configurations.

If the rule Seq in Figure 1 is a rule in the transition system $C \to C$ where $C \in \mathit{Syntax} \times \mathit{State}$, then it is invalid because the conclusion of the rule has a configuration which is just an element in *Syntax* instead of $\mathit{Syntax} \times \mathit{State}$. This error is similar to using the wrong types or numbers of parameters for a function call, which is a normal error that most programming language compilers can help the programmer identify. This error is local since the form of a rules configurations compared to the definition of the transition system is local to a single rule.

1.1.2 Unused or Undefined Symbols

Other than having bad configurations, the example in Figure 1 also has the variable *sto* show up in the premise, without being defined anywhere. This type of use-without-declaration and declaration-without-use error can both be a logical mistake or the result of a typo, with a variable like *env* being mistyped as *evn*. Having unused variables is not necessarily an error, as the use might leave the names in to provide more clarity for the reader. The bindings of variables are local to each rule, and as such this is a local error.

1.1.3 Unreachable Premises

Since the typo making *env* become *evn* in the Seq rule in Figure 1, there is no connection between the $evn' = evn[a \mapsto 1]$ premise and the initial or final configuration of the conclusion. This means the premise is not reachable which makes it "dead code," which might cause confusion for people reading the rules, and could indicate an error in the rules design. It is a local error since it is confined to a single rule.

1.1.4 Cyclic Rules

Rules can be recursive in two ways, either the premises of a rule matches itself or the final configuration of a transition references the initial configuration.

$$\begin{array}{ll}
\textsc{Factorial-RT}: & \texttt{fact}\, n \to n * \texttt{fact}\, n' \\
\textsc{Factorial-RP}: & \dfrac{\texttt{fact}\,(n-1) \to n'}{\texttt{fact}\, n \to n * n'}
\end{array}$$

Figure 2: Rule displaying rewrite both types of broken recursion.

The two rules in Figure 2 shows both recursion through premises in Factorial-RP and recursion through transitions in Factorial-RT. Transition recursion is very similar to rewrite recursion that can happen when using rewriting semantics. These definitions are ill-defined because they are recursive without any branching behavior to handle the case where n is 0. A proper implementation of the factorial function in SOS could be defined with a rule to handle the general case and a rule to handle the $n = 0$ case.

$$\begin{array}{ll}
\textsc{Factorial-RT-1}: & \dfrac{n > 0}{\texttt{fact}\, n \to n * \texttt{fact}\, n'} \\
\textsc{Factorial-RT-2}: & \texttt{fact}\, 0 \to 1
\end{array}$$

$$\begin{array}{ll}
\textsc{Factorial-RP-1}: & \dfrac{\texttt{fact}\,(n-1) \to n'}{\texttt{fact}\, n \to n * n'} \\
\textsc{Factorial-RP-2}: & \texttt{fact}\, 0 \to 0
\end{array}$$

Figure 3: Rules with the proper semantics for factorial with recursion through transitions and premises.

These examples are all based on recursion which is just one way of cyclic transitions, which in some way can be seen as a local error if the two rules are connected because of the naming. However, there are more subtle ways in which rules can be cyclic, take for example the following two rules.

$$\textsc{Reify:}\quad \frac{n' = n + 1}{\texttt{Incr}\, n \to n'}$$

$$\textsc{Incrementalize:}\quad \frac{n > 0 \quad n' = n - 1}{n \to \texttt{Incr}\, n'}$$

Figure 4: Cyclic transitions in rules that on their own makes sense.

The INCREMENTALIZE rule in Figure 4 turns a natural number into a series of increments which can make it easier to pattern-match, whereas REIFY turns an increment-expression into a number which is useful for efficient representation. Both rules might make sense individually for different purposes, but when they are both present they can form an infinite sequence of transitions (REIFY → INCREMENTALIZE → REIFY → ...).

In the rules given int the big-step style as seen with FACTORIAL-RP in Figure 2, no transition can ever take place because the premises never being fulfillable because of an infinite derivation tree. This means the semantics are ill-defined. However with small-step style semantics like FACTORIAL-RT, many transitions will take place and such infinite transition sequences can be useful if the transitions produce some effect. It is therefore harder to judge if cyclic transitions are errors in some cases, but they should be seen as a global error when they occur.

1.1.5 Permanent Intermediary Terms

When working with bindings and scoping rules, it can sometimes be useful to wrap parts of the syntax in an environment using constructs similar to the idea of Evaluation Context introduced by Felleisen and Hieb(Felleisen and Hieb 1992). These types of constructions are also called “run-time syntax” to differentiate them form the syntax that are part of the input program. However, it can be all too easy to forget defining constructions for tearing these constructs down once they can be set up.

$$\textsc{Add-Context:}\quad \langle f(v), E\rangle \to \langle f.E[v], E\rangle$$

$$\textsc{Sub-Context:}\quad \frac{\langle e, E\rangle \to \langle e', E'\rangle}{\langle f.E[e], E_s\rangle \to \langle f.E'[e'], E_s\rangle}$$

Figure 5: Rules defining and using evaluation contexts.

If we consider the small example in Figure 5, we can see that the ADD-CONTEXT rule can add a context and the SUB-CONTEXT can evaluate expressions using the context, but a rule is missing to unwrap the context after evaluation is finished. This type of error is relevant in specifications that contain such intermediary constructions that are supposed to be torn down again, and that makes this a global error since it only makes sense in the context of the failure of multiple rules to dispose of a temporary construct.

1.1.6 Unreachable Rules

Unreachability of rules, in this case, refers to specification where a transition system or a specific rule is marked as the intended "starting point" of the specification, and where rules can be judged unreachable if no possible derivation-tree or transition-sequence can lead to a rule being utilised as a premise or a base transition.

$$\textsc{Not}: \quad \frac{b \overset{b}{\Rightarrow} \texttt{true}}{\texttt{not}\, b \overset{b}{\Rightarrow} \texttt{false}}$$

$$\textsc{Const}: \quad \frac{e \overset{e}{\Rightarrow} i}{e\,\texttt{++} \overset{e}{\Rightarrow} i + 1}$$

$$\textsc{Print}: \quad \frac{e \overset{e}{\Rightarrow} i}{\texttt{print}\, e \overset{s}{\Rightarrow} i}$$

Figure 6: Very simple specification with just three rules.

If we let Print be the intended "starting point" of the specification in Figure 6, then we can see that Const is possibly reachable by virtue of a matching transition system and no restrictions in the premise of Print stop the rule from being reachable. However the Not rule is not reachable, since none of the reachable rules has premises referring to the transition system, and none of the final configurations in any of the reachable rules conclusions match the form of the initial configuration of the conclusion of Not.

An unreachable rule can be a rule from a previous iteration of the specification that is no longer needed, or it could be a rule that is intended to be reachable but some rule is missing which is supposed to connect the unreachable rule to the rest of the rules. Reachability of rules from a starting rule requires knowledge of all rules and is inherently a global error.

1.1.7 Invalid Syntax

A simple error that can happen when restructuring the specification or when someone inexperienced works with SOS, is designing rules that do not follow the syntax given by the syntactical categories.

$$G = \texttt{a} \mid \texttt{b}$$
$$C : \texttt{c} \rightarrow \texttt{a}$$

Figure 7: Specification where the rule will never apply.

The very simple example in Figure 7 shows a rule using a syntax element that is not defined anywhere. Since one of the main ideas of SOS is that the rules are syntax-directed(Plotkin 2004), these errors are trivial, but do occur and makes the specification completely ill-defined. This is a simple local error, and the most trivial of the errors that will be discussed in this section.

1.1.8 Tautological Rules

Some rules are so general that they always apply, and some so strict that they can never apply. The rules in Figure 8 are examples of tautologies with the TAUTOLOGICAL rule being of a very general form with no restrictions and the ANTI-TAUTOLOGICAL rule never being applicable, since the premise cannot be true.

$$\textsc{Tautological}: \quad \langle S, s\rangle \Rightarrow s$$

$$\textsc{Anti-Tautological}: \quad \frac{\langle S_1, s\rangle \Rightarrow s' \quad s \vdash b \Rightarrow tt \quad s \vdash \neg b \Rightarrow tt}{\langle \texttt{if}\, b\, \texttt{then}\, S_1\, \texttt{else}\, S_2, s\rangle \Rightarrow s'}$$

Figure 8: Two tautological rules.

The TAUTOLOGICAL rule not having any premises is not out of the ordinary, but this type of rule sometimes happens up when a newcomer to SOS sees a rule like PRINT from Figure 6 having a premise of the form $e \overset{e}{\Rightarrow} i$ and then coming to the conclusion that some rule must have the exact same form. This leads to rules that can be applied instead of all the other rules in the $\overset{e}{\Rightarrow}$ transition system, sometimes ruining the intended semantics.

An error like the ANTI-TAUTOLOGICAL rule can happen because of simple typos, where one of the s references was supposed to be s' instead, or something more fundamentally flawed in the logic of the premises. We assume that noone purposefully adds rules to the semantics that should never be reached, so it must be assumed to always be an error.

Since the tautology errors can be (anti-)tautological because of other rules and because they might only be errors because they ruins the semantics of other rules as is the case with the TAUTOLOGICAL rule, we should classify these errors as global errors.

1.2 Preventing errors through program analysis

Now that subsection 1.1 have given us an overview of some errors that are summed up in the table in Figure 9.

Local	Global
Incorrect Configurations	Cyclic Rules
Unused or Undefined Symbols	Unreachable Rules
Unreachable Premises	Tautological Rules
	Permanent Terms

Figure 9: Overview over the types of errors discussed.

Given that we know these errors can occur in operational semantics, it is worth asking if these problems can be found in an automated way. Analysis of programming languages and programs are some of the core exercises of computer science, but following Rice's Theorem we know that many properties of programs are not decidable in the general case by any algorithm. So we need to think about when our algorithms are over- and under-approximative about the errors we want to find by analysis.

To look at the possible analysis methods available for finding the errors mentioned in subsection 1.1 we split analysis into static and dynamic analysis.

1.2.1 Dynamic analysis

Dynamic error detection is performed during or after the evaluation of a program. This means that some input program is used to analyse the semantics of the language, and because a specific input is used, it is harder to guarantee that the results of the analysis holds for all inputs making it approximative.

Property-based testing can be seen as a form of dynamic error detection. It generates some inputs to evaluate with the defined language, and checks whether some property is upheld for the inputs, and it attempts to generate enough inputs, and without enough variety, to give confidence that the property is upheld in general, but the property does not get proven.

One prerequisite for dynamic analysis of semantics, is that an evaluator must exist for the semantics and be able to evaluate the inputs needed for analysis. In order to make property-based testing practical, the evaluator must be efficient enough so that many different inputs can be tried in an attempt to reach a certain level of certainty in the approximative analysis result.

1.2.2 Static analysis

While dynamic analysis can be used to prove that some property is upheld by the language for a given input, static analysis is not based on a given input, but analysis about the structure of the semantics in general. Since static analyses do not use specific inputs, it is easier to have results that are not approximative.

Type safety is perhaps the most popular example of static analysis. Types can be used to restrict the domain of a value which can be useful for programmers to enforce properties of the program, but can also be used by the compiler for generating better code.

1.3 A Compiler for Semantics

Creating a unified tool to perform all the analyses needed to find the mentioned errors could be done with a program that in many ways resemble a compiler. Instead of a compiler taking a programming language describing a program as input, this tool could take some meta-language describing a semantic specification as input instead.

$$\begin{tikzcd}
 & & \text{Experiment} \\
\text{Parse} \arrow[r] & \text{Analyse} \arrow[ur] \arrow[dr] & \\
 & & \text{Output Visual}
\end{tikzcd}$$

Figure 10: Simple overview over a conceptual tool for semantics.

This tool could be designed with a three-step architecture similar to a normal compiler, where the input language is parsed, analysed and then compiled. However instead of compiling it to executable code, the last step could allow the semantics definer to experiment on the semantic specification they have written. These experiments could be querying how rules fit against a tree of syntax or testing a rule to see if there are rules in the specification that satisfies the premises.

Instead of the final step being experimentation, it could also be compilation to a visual representation. This could be graphic or a mathematical typesetting in LaTeX like was done in LETOS, which in many ways shared this idea of a compiler for semantics, though focusing more on executability than analysis(Hartel 1999). The benefit of compiling to a typeset mathematical solution, is that no errors are introduced in the process of transferring the specification from the tool-readable format to the typesetting format.

To be as useful as possible for the user we can specify some goals that a tool for semantics should satisfy in order to help the user:

1. **Familiar** - Specifying semantics in the tool should be similar to the way SOS is typically taught and used in research so as to quickly let users use the tool without having to learn a new unfamiliar language.
2. **Corrective** - The tool should help the user find errors in the specifications and guide them towards a solution to problems with the semantics.
3. **Single-Source** - The user should not have to write the specification multiple times, as this can introduce errors that the tool cannot find, and having semantics that have been through mechanised error-detection be filled with errors in a transcription step would be counter-productive.

2 SOS Specifications as Graphs

To create a tool like the one proposed in subsection 1.3, we will first need to have an idea about which types of analyses are needed and possible in order to find the errors mentioned in subsection 1.1. When we talk about representing semantics with graphs, we are really talking about two types of representation, namely the flow of variables inside a single rule and the possibility of a rule leading to another rule.

2.1 Intra-Rule Graphs

Definitions of structural operational semantics usually contain a myriad of rules written mostly either in the big-step or small-step style. Rules in the big-step style often contain multiple premises, that are all required for the rule to apply and need to be considered when working with the rule. These rules often have an internal ordering of the premises, which is not based on positions or clear markings, but based on which variables are used in the configurations and expressions of the premises. Though the variables direct the ordering, not all premises are ordered since some premises might not use variables in a way that requires any ordering.

$$\textsc{Add}: \quad \frac{\langle e_1, env\rangle \Rightarrow i_1 \quad \langle e_2, env\rangle \Rightarrow i_2 \quad i = i_1 + i_2}{\langle e_1 + e_2, env\rangle \Rightarrow i}$$

Figure 11: Add encoded in the big-step style.

The rule Add defined in Figure 11 shows premises that do not have an ordering since neither of the transitions resulting in i_1 and i_2 change the environment or produce any variables used by the other, so no ordering is inherent in the definition. This means the configurations and premises of ADD only have a partial ordering. The equality premise uses the variables of the two transition premises, so in actual implementation and execution of this rule, the equality premise would be evaluated after the transition premises and the premise can therefore be seen as being ordered after the transition premises.

To give a formal definition of this graph: Let R be a rule where $\mathit{initial}(R)$ is the initial configuration of the conclusion of R and $\mathit{final}(R)$ is the final configuration of the conclusion of R and P is the set of premises of R. Then G_R is the directed intra-rule graph for R of the form $(N(R), E)$, where $N(R)$ is the set of nodes defined as $\{\mathit{initial}(R), \mathit{final}(R)\} \cup P$ and E is the set of edges on the form (n_1, v, n_2) where $n_1, n_2 \in N(R)$ and v is a variable mentioned in both n_1 and n_2.

$$\begin{tikzcd}
 & \boxed{\langle e_1, env\rangle \Rightarrow i_1} \arrow[dr, "\{i_1\}"] & & \\
\langle e_1 + e_2, env\rangle \arrow[ur, "\{e_1, env\}"] \arrow[dr, "\{e_2, env\}"'] & & \boxed{i = i_1 + i_2} \arrow[r, "\{i\}"] & i \\
 & \boxed{\langle e_2, env\rangle \Rightarrow i_2} \arrow[ur, "\{i_2\}"'] & &
\end{tikzcd}$$

Figure 12: ADD encoded as a graph. (Dashed nodes are $\mathit{initial}(R)$ and $\mathit{final}(R)$)

The result of turning the ADD rule into an intra-rule graph can be visualised as shown in Figure 12. Though the edges in the definition only have a single variable, the visual reprentation have all edges going from one node to another be shown as one, with a set of variables showing each connection.

2.1.1 Benefits of Graphs

Other than revealing unused variables, there are other scenarios where having a graph encoding allows the use of existing algorithms to reveal problems in the definition.

$$\text{BADD:}\quad \frac{(e_1, env') \Rightarrow (i_1, env'') \quad (e_2, env'') \Rightarrow (i_2, env') \quad i = i_1 + i_2}{(e_1 + e_2, env) \Rightarrow (e_3, env')}$$

Figure 13: BADD encoded in the big-step style.

The above BADD rule in Figure 13 contains multiple errors, but these can be hard to spot when reading the rule. Finding the errors programmatically would be beneficial, but without a fitting data structure it can be harder to spot a viable algorithmic solution immediately. This is another case where working on a graph representation could prove beneficial compared to the normal unordered definition.

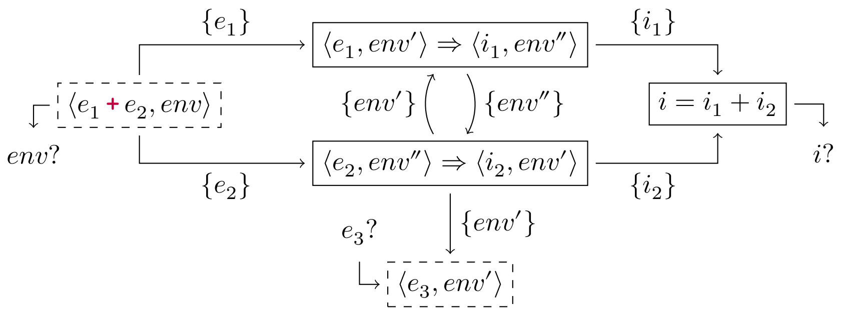

$\{e_1\}$
$\langle e_1, env' \rangle \Rightarrow \langle i_1, env'' \rangle$
$\{i_1\}$
$\langle e_1 + e_2, env \rangle$
$\{env'\}$
$\{env''\}$
$i = i_1 + i_2$
$env?$
$\langle e_2, env'' \rangle \Rightarrow \langle i_2, env' \rangle$
$i?$
$\{e_2\}$
$\{i_2\}$
$e_3?$
$\{env'\}$
$\langle e_3, env' \rangle$

Figure 14: BAdd encoded in a graph.

When looking at the graph encoding in Figure 14, some errors might become more apparent immediately, both to manual error-checking but also possible to automated search for errors.

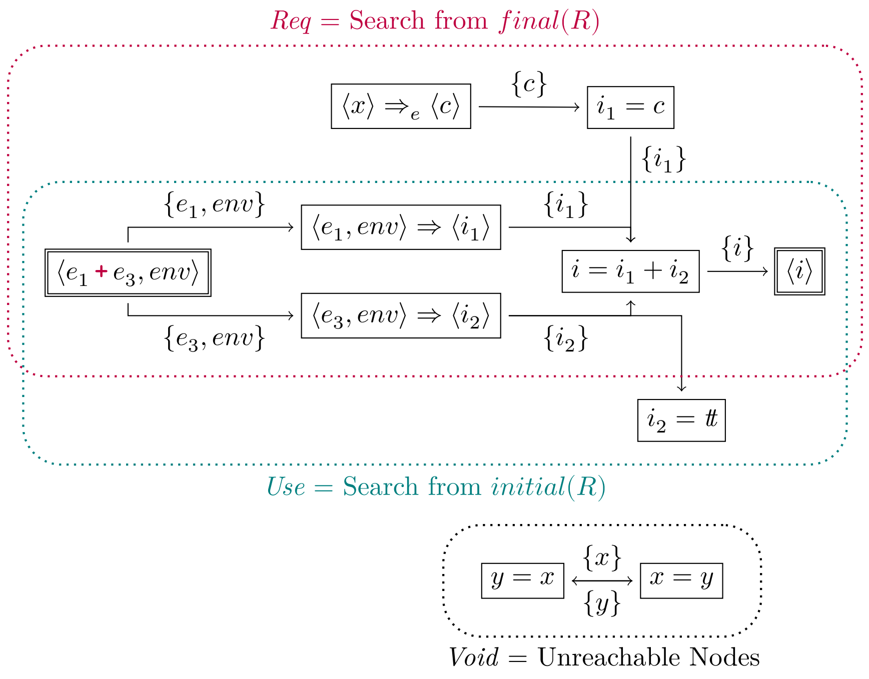
The first error is the variable *env* being defined but never used. This is an error in this rule and might be caused by the user mistyping the name of the variable in one of the premises. It is not always an error to have unused variables, because names can be left for clarity. The equality premise defines i but the variable is not used, and there is no real outgoing edges from the node, making it a dead end. Being a dead end is not a problem in the case of the premise being a constraint (like $a = \textbf{True}$). The conclusion needs a c variable defined, but it is not defined so the conclusion is nonsensical.

One of the big benefits of the graph representation, is that spotting cyclic dependencies becomes very easy with any graph algorithm that checks for cycles. An example of this is the two transition-premises in BAdd that are mutually dependent, thereby creating a cycle in the graph.

In many ways, making sure that the intra-rule graph is a directed acyclic graph, such that we can make a topological ordering of all the nodes. Having a topological ordering is also a valid evaluation order of the different nodes, such that the bindings each expression needs is supplied before evaluation.

2.1.2 Building the Graph

While the graph seems to provide good opportunities for analysis, the analysis cannot begin before we have a way of building the flow graph. Because we do not necessarily know which premise defines a variable, we cannot simply connect all definitions to uses. We can start building the graph based on required variables, starting from the concluding configuration. Alternatively we can start the analysis from the initial configuration, exploring all premises that are possible with the currently defined variables.



Req = Search from *final*(*R*)
$\langle x\rangle \Rightarrow_e \langle c\rangle$
$\{c\}$
$i_1 = c$
$\{i_1\}$
$\{e_1, env\}$
$\langle e_1, env\rangle \Rightarrow \langle i_1\rangle$
$\{i_1\}$
$\langle e_1 + e_3, env\rangle$
$i = i_1 + i_2$
$\{i\}$
$\langle i\rangle$
$\langle e_3, env\rangle \Rightarrow \langle i_2\rangle$
$\{e_3, env\}$
$\{i_2\}$
$i_2 = tt$
Use = Search from *initial*(*R*)
$y = x$
$\{x\}$
$\{y\}$
$x = y$
Void = Unreachable Nodes

Figure 15: The two different searches results in different sets of nodes.

Doing both of the searches shown in Figure 15 gives us three sets of nodes.

1. *Req* is the set of nodes that provide variables needed for the conclusion of the rule, found by the backwards search.
2. *Use* is the set of nodes that use variables given in the initial premise of the rule, found by the forwards search.
3. *Void* is the set of nodes that are in the graph, but is in neither of the other sets.

These sets are interesting, because doing the search to create them reveals missing and unused variables, and because the *Void* set is premises that are not reachable in the rule, and therefore should not logically exist as they are.

2.1.3 Breadth-first Search

Both of the searches can use the same algorithm to search for variables. Given a rule R we have the set of nodes $N(R)$ where $n \in N(R)$, we can have $reqs(n)$ be the set of variables mentioned in n that are used but not defined in n and $prov(n)$ be the set of variables that are defined in n.

```
findproviders(source_node) =
    nodes <- empty set
    edges <- empty set
    for each variable v in reqs(source_node)
        providing_node <- none
        for each node n in N(R)
            if v is in pros(n)
                if node = none
                    providing_node <- n
                else
                    throw DoubleDefinitionError
        if node = none
            throw NoDefinitionError
        else
            nodes <- nodes U { providing_node }
            edges <- edges U {
                (source_node, v, providing_node)
            }
    return (nodes, edges)
```

Figure 16: The `findproviders` algorithm that either builds a graph of the source node and all nodes that provide variables, or exits with a bind error.

```
dfs(find, seen_nodes, edges, node_stack) =
    if node_stack is empty then
        return (seen_nodes, edges)
    else
        n <- head of node_stack
        tail <- tail of node_stack
        (p,e) <- find(n)
        unseen_nodes <- p \ seen_nodes
        new_node_stack <- tail + unseen_nodes
        return dfs(
            find,
            seen_nodes U {n},
            edges U e,
            new_node_stack
        )
```

Figure 17: The BFS algorithm used to search backwards through nodes.

Given a rule R, and definitions of $reqs(n)$ and $prov(n)$, the searching algorithm will return a graph with the nodes required to reach the conclusion by calling it as $dfs(findproviders, \emptyset, \emptyset, [final(R)])$ which we can call dfs_{back}. The algorithm shown in Figure 17 also works for the forwards search from $initial(R)$, if the method `findreceivers` is defined as `findproviders` from Figure 16 but with the definition of $reqs(n)$ and $prov(n)$ swapped. Thus we have dfs_{fore} defined as

$dfs(findreceivers, \emptyset, \emptyset, [initial(R)])$ which means we can build both the graph showing both the edges needed to get to the final configuration, and the edges reachable from the initial configuration. When we have both of these graphs, we can find the unreachable nodes, by combining the nodes from either graph, so $void_R$ is $N(R) \setminus N(dfs_{back}) \cup N(dfs_{fore})$.

2.2 Inter-Rule Graphs

The other type of graph is one showing the connections between different rules. This graph is also a directed graph similar to the intra-rule graph, but the nodes are individual rules and the edges are the connections between the rules caused by transition premises.

$$\overset{e}{\Rightarrow} \in \mathit{Syntax} \times \mathit{Integer}$$
$$\overset{b}{\Rightarrow} \in \mathit{Syntax} \times \mathit{Boolean}$$
$$\overset{s}{\Rightarrow} \in (\mathit{Syntax} \times \mathit{Environment}) \times \mathit{Environment}$$
$$e = e \,\texttt{+}\, e \mid e \,\texttt{++}$$
$$b = b \,\texttt{=}\, b \mid \texttt{true} \mid \texttt{false}$$
$$s = \texttt{while}\, b\, \texttt{do}\, s \mid \texttt{skip}$$

$$\textsc{Add-One:}\quad \frac{e \overset{e}{\Rightarrow} i \quad i' = i + 1}{e \,\texttt{++} \overset{e}{\Rightarrow} i'}$$

$$\textsc{Add:}\quad \frac{e_1 \overset{e}{\Rightarrow} i_1 \quad e_2 \overset{e}{\Rightarrow} i_2 \quad i' = i_1 + i_2}{e_1 \,\texttt{+}\, e_2 \overset{e}{\Rightarrow} i'}$$

$$\textsc{Eq-True:}\quad \frac{e_1 \overset{e}{\Rightarrow} i_1 \quad e_2 \overset{e}{\Rightarrow} i_2 \quad i_1 = i_2}{e_1 \,\texttt{=}\, e_2 \overset{b}{\Rightarrow} \mathit{tt}}$$

$$\textsc{While-True:}\quad \frac{b \overset{b}{\Rightarrow} tt \quad (S, env) \overset{s}{\Rightarrow} env' \quad (\texttt{while}\, b\, \texttt{do}\, S, env') \overset{s}{\Rightarrow} env''}{(\texttt{while}\, b\, \texttt{do}\, S, env) \overset{s}{\Rightarrow} env''}$$

Figure 18: A full specification involving containing two rules.

A key difference is that in order to gain insights from the rules, we will have to limit which edges are created. Many transition premises in SOS apply to many rules, such as $e_1 \Rightarrow_e i_1$ in If-True that applies to both Add and Add-One, because it is so general that it matches every single rule for expressions. This can be seen in two ways, either because the premise uses the variable e and the two initial configurations of the rules matches the grammar, or because it uses the transition system $\Rightarrow_e$ which the two rules use. The use of the transition system is the more clear reason why the rules are connected, but looking at the actual syntax can be used when some constructs like `while` loops that often are defined in terms of themselves and `if` expressions.

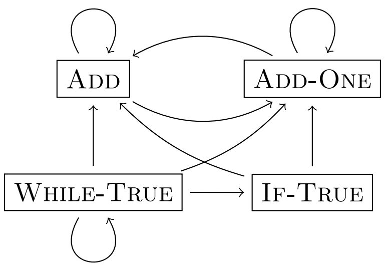


Figure 19: Graph of the specification defined in Figure 18.

The graph in Figure 19 shows the specification from Figure 18 turned into an inter-rule graph. In this graph it is not immediately obvious if there is any problems with cyclic errors like the ones discussed in subsubsection 1.1.4. The WHILE-TRUE is possibly a rule where there is a risk of an infinite derivation. This reason WHILE-TRUE is the only risky rule is because rules like ADD only reference themselves as a sub-expression, which does not have the risks that rules like WHILE do as explained in subsubsection 1.1.4.

To get more out of an inter-rule analysis, we define the inter-rule graph as having both *weak* edges that can be a sub-expression reference and *strong* edges that only allow direct grammar-matching.

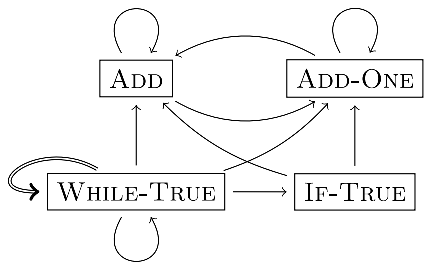


Figure 20: Graph of the specification defined in Figure 18, with strong edges shown in bold.

Given the specification S then R is the set of rules in S and G_S is the inter-rule graph of S of the form (R, E_s, E_w) We define E_s as the set of edges on the form (r_1, r_2) where $r_1, r_2 \in R$ and $strongmatch(r_1, r_2)$ holds. We define E_w in the same way as E_s except $weakmatch(r_1, r_2)$ has to hold instead.

In the context of the specification given in Figure 18, we say that $\langle e \rangle \overset{e}{\Rightarrow}$ ... is a category-matching transition while the transition $\langle e \, \texttt{++} \rangle \overset{e}{\Rightarrow}$ ... is a production-matching transition. Using this, we say that the property $strongmatch(r_1, r_2)$ holds if any of the production-matching transition premises r_1 matches the conclusion of r_2. This can be seen in the way the the third premise of the WHILE-TRUE rule matches the conclusion of WHILE-TRUE itself, meaning $strongmatch(\text{WHILE-TRUE}, \text{WHILE-TRUE})$ holds. The property $weakmatch(r_1, r_2)$ holds if any of the category-matching transition premises in r_1 matches the conclusion of r_2. We can see that $weakmatch(\text{IF-TRUE}, \text{ADD})$ holds, because both the first and second premise of IF-TRUE is a category-matching transition premise that matches the conclusion of ADD.

3 A Type System for SOS

If a language is created to specify semantics for the tool proposed in subsection 1.3, then it opens up for one of the most useful techniques for enforcing properties about languages, namely static type checking. Static type checking is a formal verification technique that is used to enforce certain correctness properties of programs in a language by creating a type system and excluding all programs, which are not well-typed under this type system, from the set of valid programs.

A type system comprises of a set of rules that associates the constructs in a language with a type, and a program is then considered valid only if types associated with the constructs are the ones expected in the context of the construct. By checking types it should be possible to reduce the set of specifications that can be defined using the meta-language to a subset that excludes many nonsense specifications from being considered valid. The meta-language for defining operational semantics should have a type system, that resembles the types in normal SOS specifications, namely a type system based around syntactical categories.

3.1 Preventing Errors in SOS via Types

To understand which of the errors mentioned in subsection 1.1 we will be able to find using type-checking, we first need to understand the type system of normal SOS specifications.

3.1.1 Variables

The variables in the inference-rules of SOS specification are often on the form e_1' where e is defined to be a meta-variable in a syntactical category. The definitions of the syntactical categories are important to the SOS specifications, and are often done by direct definition of a set like $\{i \mid i \in \mathbb{N}$ and $i > 0\}$ or by specifying the formation rules that define the elements of the set(Hüttel 2010).

$$S = (\mathbf{Expr}, \overset{e}{\Rightarrow}, \mathbf{Integer}) \text{ where}$$
$$=> e \;\subset \mathbf{Expr} \times \mathbf{Integer}$$
$$i \in \mathbf{Integer} \qquad \mathbf{Integer} = \mathbb{Z} \qquad e \in \mathbf{Expr} \qquad e ::::= e + e \mid i$$
$$\textsc{Add}: \quad \frac{e_1 \overset{e}{\Rightarrow} i_1 \quad e_2 \overset{e}{\Rightarrow} i_2 \quad i = i_1 + i_2}{e_1 + e_2 \Rightarrow i}$$

Figure 21: A simple semantic specification for the transition system S done as SOS.

The use of meta-variables and variables can be illustrated with the specification defined in Figure 21 which shows variables both of a described syntactical category (the meta-variable i) and a category defined via formation rules (the meta-variable e). This inclusion of the meta-variables in the naming of the variables can be seen as a form of explicitly declaring the types of variables, which is also commonly seen in languages like C where variables are defined by a type annotation like `int a` or Fortran where the type of a variable can be dependant on the first letter of the variable name, such as `i1` being an integer. If this naming scheme is used in the meta-language, then the type of a variable can be found by looking up the name in the meta-variables defined in the specification.

3.1.2 Transitions and Configurations

Similarly to the variable types, Figure 21 also shows the transition system S with the transition relation defining the left-hand configuration as being an element in **Expr** and the right-hand configuration being an element in **Integer**. This means a transition such as $i_1 \overset{e}{\Rightarrow} i_2$ is invalid, as the element (i_1, i_2) is not in the set $\mathbf{Expr} \times \mathbf{Integer}$, and if we can find this via type-checking then we could detect a whole class of errors related to configuration of the wrong form.

Furthermore, because we are given the definition of the transition relation $\overset{e}{\Rightarrow}$, we are able to infer the variables a and b when given a transition on the form $a \overset{e}{\Rightarrow} b$ which we could use to avoid the need for explicit type annotations of a and b in this case.

3.1.3 Constraint and Definition Premises

Many rules have premises that consists of expressions other than just the requireming a certain transition to be possible. This can be seen in the example by looking at the third premise of the Add rule, which uses an equality as a further requirement. In this case the equality also acts as the definition of the i variable, but cases such as $i < 4$ is also sometimes used to constrain the rules further. Since the constraint has to hold for the rule to hold, we could require that cases like an equality-check has both sides of the expression be of identical types, which would not only allow us to rule out premises such as $i_1 = e_2$ that will never hold, but also to infer the value of a in a premise such as $i_1 = a$ to be the same as i_1.

3.1.4 Type Inference Across a Rule

When we think of premises such as $a = b$ and $b = 1$, it might be intuitive that a an integer because b is, but this is not as simple when defining an algorithm to perform type-checking. If a naive sequential type-check is performed on the sequence of expressions $(a = b) \to (b = 1)$, we might already encounter an error upon checking $a = b$ because the type-checker has no idea the type of either variable, since it has not checked $b = 1$ yet. We can modify the naive algorithm to still allow infering the type of a by either reordering the expressions or by using typevariables for variables until they are inferable.

We can easily reorder the $(a = b) \to (b = 1)$ sequence, but if we are given the sequence $(a = b) \to (b = c) \to (c = a)$ and told that b is an integer, it can be hard to know which order the variables should be checked in to come to the conclusion that a, b and c all are of the same type (and same value as well).

One idea to avoid reordering is to have any unknown variable be of a type that is a super-type for all types such as what TypeScript does with the *any* type. The benefit of this is that it is simple and efficient to implement, but the drawback that type information does not get propagated backwards and would this let $(a = b) \to (a = 1) \to (b = \emptyset)$ be correct according to the type-checker.

A more sound alternative is to give every variable with no inferable type a unique type-variable and have inference generate constraints, also called equalities, for these type-variables that can be used to propagate type information backwards. This would mean type-checking a sequence like $(a = b) \to (b = 1)$ would first infer a to be of the unknown type $\ x$ and b to be of type $\ y$ with the constraint that $\ x = \ y$, and then add another constraint that $\ y = \mathbb{Z}$. A final step could then collapse it the constraints so we get that a and b both are of the type $\mathbb{Z}$. To simplify, we can see it as immediate back-propagation in the simple example, so that type-checking $(a = b) \to (b = 1)$

will infer a and b to be of type x, and then in the second step we can infer that b must be $\mathbb{Z}$, but it is already x so we perform substitution of x with $\mathbb{Z}$.

This method of inference is known as unification and is the type inference method we will be using because of the benefits just mentioned. Unification is a algorithm for solving equations between symbolic terms that can be used in type inference algorithms to create a uni-directional type inference. Unification-based type inference is useful for infering types when the code in such a way that variables can be used before being declared lexically, and since rule premises are supposed to be an unordered set, this kind of type inference seemed a natural fit.

3.2 Type Rules

We have derived the type rules of the language from the implementation and present them as inference rules. Since these inference rules are syntax-oriented, they can only be understood with the context of the grammar, so we include also the grammar in Figure 22.

$$
\begin{aligned}
syn &\in Syntax \\
\alpha &\in TypeVars \\
x &\in Identifiers \\
arr &\in Arrows \\
p &\in PrimitiveType \\
\\
cnf &::= \langle (cnfs\, \texttt{,}\,)...cnfs \rangle \\
cnfs &::= cel...cel \\
tr &::= cnf\ arr\ cnf \\
prem &::= tr \mid e \mid e\, \texttt{:=}\, e \\
cel &::= syn \mid xe \mid \texttt{(}\, cnfs\, \texttt{)} \\
cat &::= \texttt{category}\ x\ \texttt{=}\ T \\
sys &::= \texttt{system}\ T\ arr\ T \\
meta &::= \texttt{meta}\ x\ \texttt{=}\ T \\
rule &::= \texttt{rule}\ x\ prem...prem\, \texttt{---}\, tr \\
S &::= cat...cat\, \texttt{,}\ sys...sys\, \texttt{,}\ meta...meta\, \texttt{,}\ rule...rule
\end{aligned}
\qquad
\begin{aligned}
T &::= \alpha \\
&\mid p \\
&\mid syn \\
&\mid (T\ \texttt{x})...T \\
&\mid (T\, \texttt{U})...T \\
&\mid T \rightarrow T \\
xe &::= x \\
&\mid T\, \texttt{\_}\, x \\
&\mid x\, \texttt{:}\, T \\
&\mid xe\, \texttt{[}\, x \mapsto e\, \texttt{]} \\
e &::= xe \\
&\mid e\, \texttt{(}\, (e\, \texttt{,}\,)...e\, \texttt{)} \\
&\mid e\, \texttt{=}\, e \\
&\mid e\ \texttt{!=}\, e
\end{aligned}
$$

Figure 22: A grammar of the Semantec language.

The type environment E is a partial function from identifiers or arrows to types. It is used to store information about bindings and allow us to pass this information around in a single object.

$$\mathrm{E} : (Identifiers \cup Arrows) \rightharpoonup Types$$

typeVars is a function for getting the set of all type variables in the environment, which is useful for creating fresh type variables.

$$typeVars(E) = \bigcup_{T \in ran(E)} tv(T)$$

$$tv(t) = \begin{cases} \{\alpha\} & \textit{if } t = \alpha \\ tv(T_1) \cup ... \cup tv(T_n) & \textit{if } t = (T_1 \texttt{ x })...T_n \\ tv(T_1) \cup ... \cup tv(T_n) & \textit{if } t = (T_1 \texttt{ U })...T_n \\ tv(T_1) \cup tv(T_2) & \textit{if } t = T_1 \rightarrow T_2 \\ \emptyset & \textit{otherwise} \end{cases}$$

Σ denotes a substitution function, which is a mapping from type variables to types. It is used for substituting type variables after constraints have been made on them, and to do this we have some helper functions. $subst_T$ is used for recursively applying Σ to a type. $subst_E$ and $subst2_E$ are used for recursively applying Σ to the codomain of the type environment.

$$\Sigma : \alpha \rightharpoonup T$$

$$subst_E(\Sigma, E) = subst2_E(\Sigma, E, \emptyset)$$

$$subst2_E(\Sigma, E_{in}, E_{out}) = \begin{cases} subst2_E(\Sigma, E_{tmp}, E_{out}[x \mapsto subst_T(\Sigma, T)]) & \textit{if } E_{tmp}[arr \mapsto T] = E_{in} \\ subst2_E(\Sigma, E_{tmp}, E_{out}[x \mapsto T]) & \textit{if } E_{tmp}[arr \mapsto T] = E_{in} \\ E_{out} & \textit{otherwise} \end{cases}$$

$$subst_T(\Sigma, T) = \begin{cases} T_2 & \textit{if } \alpha = T \textit{ and } T_2 = \Sigma(\alpha) \\ (subst_T(\Sigma, T_1) \texttt{ x })...subst_T(\Sigma, T_n) & \textit{if } (T_1 \texttt{ x })...T_n = T \\ (subst_T(\Sigma, T_1) \texttt{ U })...subst_T(\Sigma, T_n) & \textit{if } (T_1 \texttt{ U })...T_n = T \\ T & \textit{otherwise} \end{cases}$$

normalize is used to normalize the substitutions by recursively applying Σ to itself until a fixed point is reached.

$$normalize(\Sigma_1) = \begin{cases} norm(\Sigma_1, \Sigma_1, \emptyset) & \textit{if } \Sigma_1 \neq norm(\Sigma_1, \Sigma_1, \emptyset) \\ \Sigma_1 & \textit{otherwise} \end{cases}$$

$$norm(\Sigma_{pre}, \Sigma_{in}, \Sigma_{out}) = \begin{cases} norm(\Sigma_{pre}, \Sigma_{tmp}, \Sigma_{out}[\alpha \mapsto subst_T(\Sigma_{pre}, T)]) & \textit{if } \Sigma_{in} = \Sigma_{tmp}[\alpha \mapsto T] \\ \Sigma_{out} & \textit{otherwise} \end{cases}$$

The type rules use the four judgement forms listed below.

$$\begin{array}{rl}
\textsc{Decl}: & E \vdash_D \mathit{any} \dashv E \\
\textsc{Check}: & E \vdash_C \mathit{any} \dashv E, \Sigma \\
\textsc{Infer}: & E \vdash_R \mathit{any} :: T \dashv E, \Sigma \\
\textsc{Unify}: & E, \Sigma \vdash_U T\ T \dashv \Sigma
\end{array}$$

Decl is used for updating the environment with declarations. Check is used for type-checking rules. Infer is used for infering the types of expressions and configurations. Unify is used for unifying types by generating type substitutions.

The $\emptyset$ symbol is used to denote empty functions.

The $E_{n...m}$ form is used for concatenating multiple environments into one, which.

$$\textsc{Spec}: \quad \frac{\begin{array}{l}
E_0^n = \emptyset \\
(E_1^n \vdash_D \mathit{cat}_1 \dashv E_1^n)...(E_{n-1}^n \vdash_D \mathit{cat}_n \dashv E_n^n) \\
E_0^m = E_n^n \\
(E_0^m \vdash_D \mathit{sys}_1 \dashv E_1^m)...(E_{m-1}^m \vdash_D \mathit{sys}_m \dashv E_m^m) \\
E_0^o = E_m^m \\
(E_0^o \vdash_D \mathit{meta}_1 \dashv E_1^o)...(E_{o-1}^o \vdash_D \mathit{meta}_o \dashv E_o^o) \\
(E_o^o \vdash_C \mathit{rule}_1 \dashv E_1^p, \Sigma_1)...(E_o^o \vdash_C \mathit{rule}_p \dashv E_p^p, \Sigma_p)
\end{array}}{\mathit{cat}_1...\mathit{cat}_n \ \mathbf{,}\ \mathit{sys}_1...\mathit{sys}_m \ \mathbf{,}\ \mathit{meta}_1...\mathit{meta}_o \ \mathbf{,}\ \mathit{rule}_1...\mathit{rule}_p}$$

$$\textsc{Meta}: \quad \frac{x \notin \mathit{domain}(\mathrm{E})}{\mathrm{E} \vdash_D \texttt{meta}\, x \,\texttt{=}\, T \dashv E[x \mapsto T]}$$

$$\textsc{Cat}: \quad \frac{x \notin \mathit{domain}(\mathrm{E})}{\mathrm{E} \vdash_D \texttt{category}\, x \,\texttt{=}\, T \dashv E[x \mapsto T]}$$

$$\textsc{System}: \quad \frac{\mathit{arr} \notin \mathit{domain}(\mathrm{E})}{\mathrm{E} \vdash_D \texttt{system}\, T_1\ \mathit{arr}\ T_2 \dashv E[\mathit{arr} \mapsto (T_1 \rightarrow T_2)]}$$

$$\textsc{Rule}: \quad \frac{\begin{array}{l}
\mathrm{E}_0 \vdash_C \mathit{prem}_1 \dashv E_1, \Sigma_1 \\
... \\
\mathit{subst}_E(\Sigma_{(n-1)..1}, \mathrm{E}_{n-1}) \vdash_C \mathit{prem}_n \dashv E_n, \Sigma_n \\
\mathit{subst}_E(\Sigma_{n...1}, \mathrm{E}_n) \vdash_C \mathit{tr} \dashv E_m, \Sigma_m
\end{array}}{\mathrm{E}_0 \vdash_C \texttt{rule}\, x\ \mathit{prem}_1...\mathit{prem}_n \ \texttt{---}\ \mathit{tr} \dashv \mathrm{E}_0, \emptyset}$$

$$\textsc{Trans}: \quad \frac{\begin{array}{l}
E_0(\mathit{arr}) = T_1 \rightarrow T_2 \\
\mathrm{E}_0 \vdash_R \mathit{cnf}_1 :: T_3 \dashv E_1, \Sigma_1 \\
\mathit{subst}_E(\Sigma_1, \mathrm{E}_1) \vdash_R \mathit{cnf}_2 :: T_4 \dashv E_2, \Sigma_2 \\
T_1 = \mathit{subst}_T(\Sigma_{2...1}, T_3) \\
T_2 = \mathit{subst}_T(\Sigma_{2...1}, T_4)
\end{array}}{\mathrm{E}_0 \vdash_C \mathit{cnf}_1\, \mathit{arr}\ \mathit{cnf}_2 \dashv E_2, \Sigma}$$

Figure 23: Type rules for checking.

Figure 23 shows the top-level rules. SPEC defines the order in which top-level terms are evaluated - first *cat*, *sys*, and *meta* terms are used to update the environment, which simply adds them to the environment. Then all the rules are checked, but the checking of one rule does not affect the environment for the next rule.

RULE defines the evaluation for the components of the rule. While there is an order to this, the order does not matter because unification will ensure that the types match up regardless of order.

Similarly TRANS has an order that does not matter due to unification. The type substitutions are collected first, and then the substitutions of the types are compared for equality with the expected types.

$$\textsc{Def}: \quad \frac{\mathrm{E}_0 \vdash_R e_1 :: T \dashv E_1, \Sigma_1 \quad \mathit{subst}_E(\Sigma_1, \mathrm{E}_1) \vdash_R e_2 :: T \dashv E_2, \Sigma_2}{\mathrm{E}_0 \vdash_R e_1 \,\texttt{:=}\, e_2 :: T \dashv E_2, \Sigma_{2\ldots1}}$$

$$\textsc{Conf-1}: \quad \frac{\begin{array}{l} \mathrm{E}_0 \vdash_R \mathit{cnfs}_1 :: T_1 \dashv E_1, \Sigma_1 \\ \ldots \\ \mathit{subst}_E(\Sigma_{(n-1)\ldots1}, \mathrm{E}_{n-1}) \vdash_R \mathit{cnfs}_1 :: T_n \dashv E_n, \Sigma_n \end{array}}{\mathrm{E}_0 \vdash_R \langle(\mathit{cnfs}_1 \,\texttt{,}\,)\ldots\mathit{cnfs}_n\rangle :: (T_1 \,\texttt{x}\,)\ldots T_n \dashv E_n, \Sigma_{n\ldots1}}$$

$$\textsc{Conf-2}: \quad \frac{\begin{array}{l} \mathrm{E}_0 \vdash_R \mathit{cel}_1 :: T_1 \dashv E_1, \Sigma_1 \\ \ldots \\ \mathit{subst}_E(\Sigma_{(n-1)\ldots1}, \mathrm{E}_{n-1}) \vdash_R \mathit{cel}_1 :: T_1 \dashv E_n, \Sigma_n \end{array}}{\mathrm{E}_0 \vdash_R \mathit{cel}_1\ldots\mathit{cel}_n :: (T_1 \,\texttt{x}\,)\ldots T_n \dashv E_n, \Sigma_{n\ldots1}}$$

$$\textsc{Syn}: \quad \mathrm{E} \vdash_R \mathit{syn} :: \mathit{Syntax} \dashv E, \emptyset$$

$$\textsc{Prod}: \quad \frac{\begin{array}{l} \mathrm{E}_0 \vdash_R e_1 :: T_1 \dashv E_1, \Sigma_1 \\ \ldots \\ \mathit{subst}_E(\Sigma_{(n-1)\ldots1}, \mathrm{E}_{n-1}) \vdash_R e_n :: T_n \dashv E_n, \Sigma_n \end{array}}{\mathrm{E}_0 \vdash_R (e_1 \,\texttt{,}\,)\ldots e_2 :: (T_1 \,\texttt{x}\,)\ldots T_n \dashv E_n, \Sigma_{n\ldots1}}$$

$$\textsc{Var-1}: \quad \frac{x \notin \mathit{dom}(\mathrm{E}) \quad \alpha \notin \mathit{typeVars}(\mathrm{E})}{\mathrm{E} \vdash_R x :: \alpha \dashv E[x \mapsto \alpha], \emptyset}$$

$$\textsc{Var-2}: \quad \frac{E(x) = T}{\mathrm{E} \vdash_R x :: T \dashv E, \emptyset}$$

$$\textsc{VarT}: \quad \frac{\begin{array}{l} \mathrm{E}_0 \vdash_R x :: T_2 \dashv E_1, \Sigma_1 \\ \mathit{subst}_E(\Sigma_1, E_1) \vdash_U T_1 \; T_2 \dashv \Sigma_2 \\ \Sigma = \Sigma_{2\ldots1} \\ T = \mathit{subst}_T(\Sigma, T_1) \end{array}}{\mathrm{E}_0 \vdash_R x \,\texttt{::}\, T_1 :: T \dashv E_1[x \mapsto T], \Sigma}$$

$$\textsc{TVar}: \quad \frac{\mathrm{E}_0 \vdash_R x \,\texttt{::}\, T :: T \dashv E_1, \Sigma}{\mathrm{E}_0 \vdash_R T \,\texttt{\_}\, x :: T \dashv E_1, \Sigma}$$

$$\textsc{Bind}: \quad \frac{\begin{array}{l} E_0 \vdash_R xe :: T_{xe} \dashv E_1, \Sigma_1 \\ \mathit{subst}_E(E_1, \Sigma_1) \vdash_R x :: T_x \dashv E_2, \Sigma_2 \\ \mathit{subst}_E(E_2, \Sigma_{2\ldots1}) \vdash_R e :: T_e \dashv E_3, \Sigma_3 \\ \mathit{subst}_E(E_3, \Sigma_{3\ldots1}), \Sigma_{3\ldots1} \vdash_U T_{xe} \; (T_x \,\texttt{→}\, T_e) \dashv \Sigma_4 \end{array}}{E_0 \vdash_R xe \,\texttt{[}\, x \mapsto e \,\texttt{]}\, \dashv \mathit{subst}_E(E_3, \Sigma_{4\ldots1}), \Sigma_{4\ldots1}}$$

Figure 24: Type rules for inference part 1.

$$\textsc{Equal}: \quad \frac{\begin{array}{c} E_0 \vdash_R e_1 :: T_1 \dashv E_1, \Sigma_1 \\ E_1 \vdash_R e_2 :: T_2 \dashv E_2, \Sigma_2 \\ E_2, \Sigma_{2\ldots1} \vdash_U T_1 \; T_2 \dashv \Sigma_3 \end{array}}{E_0 \vdash_R e_1 \texttt{=} e_2 :: \mathit{subst}_T(\Sigma_{3\ldots1}, T_1) \dashv E_2, \Sigma_{3\ldots1}}$$

$$\textsc{NEqual}: \quad \frac{\begin{array}{c} E_0 \vdash_R e_1 :: T_1 \dashv E_1, \Sigma_1 \\ E_1 \vdash_R e_2 :: T_2 \dashv E_2, \Sigma_2 \\ E_2, \Sigma_{2\ldots1} \vdash_U T_1 \; T_2 \dashv \Sigma_3 \end{array}}{E_0 \vdash_R e_1 \texttt{!=} e_2 :: \mathit{subst}_T(\Sigma_{3\ldots1}, T_1) \dashv E_2, \Sigma_{3\ldots1}}$$

$$\textsc{Call}: \quad \frac{\begin{array}{l} E \vdash_R e_0 :: T_0 \dashv E_0, \Sigma_0 \\ \mathit{subst}_E(E_0, \Sigma_0) \vdash_R e_1 :: T_1 \dashv E_1, \Sigma_1 \\ \ldots \\ \mathit{subst}_E(E_n, \Sigma_{n\ldots0}) \vdash_R e_n :: T_n \dashv E_n, \Sigma_n \\ \alpha \notin \mathit{typeVars}(E) \\ \mathit{subst}_E(E_n, \Sigma_{n\ldots0}) \vdash_U T_0 \; (((T_1 \texttt{ x })\ldots T_n) \texttt{→} \alpha) \dashv \Sigma_{n+1} \\ T = \mathit{subst}_T(\Sigma_{(n+1)\ldots0}, \alpha) \end{array}}{E \vdash_R e_0 \texttt{(} (e_1 \texttt{ , })\ldots e_n \texttt{)} :: T \dashv E_n, \Sigma_{(n+1)\ldots0}}$$

Figure 25: Type rules for inference part 2.

Figure 24 defines the rules for inferring types of from terms.

$$\textsc{Unify-Prim}: \quad \frac{p_1 = p_2}{E, \Sigma \vdash_U p_1\ p_2 \dashv \Sigma}$$

$$\textsc{Unify-Prod}: \quad \frac{\begin{array}{l} E, \Sigma_0 \vdash_U T_1^a\ T_1^b \dashv \Sigma_1 \\ \ldots \\ \mathit{subst}_E(\Sigma_{(n-1)\ldots 0}, E), \Sigma_{(n-1)\ldots 0} \vdash_U T_n^a\ T_n^b \dashv \Sigma_n \\ n = m \end{array}}{E, \Sigma_0 \vdash_U ((T_1^a\ \texttt{x})\ldots T_n^a)\ ((T_1^b\ \texttt{x})\ldots T_m^b) \dashv \Sigma_{n\ldots 0}}$$

$$\textsc{Unify-Func}: \quad \frac{E, \Sigma_0 \vdash_U T_1^a\ T_1^b \dashv \Sigma_1 \quad \mathit{subst}_E(\Sigma_1, E), \Sigma_1 \vdash_U T_1^a\ T_1^b \dashv \Sigma_2}{E, \Sigma_0 \vdash_U (T_1^a \rightarrow T_2^a)\ (T_1^b \rightarrow T_2^b) \dashv \Sigma_{2\ldots 0}}$$

$$\textsc{Unify-Var}: \quad E, \Sigma \vdash_U \alpha\ T \dashv \mathit{normalize}(\Sigma[\alpha \mapsto T])$$

$$\textsc{Unify-Union}: \quad \frac{E, \Sigma_0 \vdash_U T_m^a\ T^b \dashv \Sigma_1 \quad 1 \leq m \leq n}{E, \Sigma_0 \vdash_U ((T_1^a\ \texttt{U})\ldots T_n^a)\ T^b \dashv \Sigma_{1\ldots 0}}$$

$$\textsc{Swap-Var}: \quad \frac{E, \Sigma_0 \vdash_U \alpha\ T \dashv \Sigma_1 \quad T \neq \alpha_2}{E, \Sigma_0 \vdash_U T\ \alpha_1 \dashv \Sigma_1}$$

$$\textsc{Swap-Union}: \quad \frac{E, \Sigma_0 \vdash_U ((T_1^b\ \texttt{U})\ldots T_n^b)\ T^a \dashv \Sigma_1 \quad T^a \neq ((T_1^a\ \texttt{U})\ldots T_m^a)}{E, \Sigma_0 \vdash_U T^a\ ((T_1^b\ \texttt{U})\ldots T_n^b) \dashv \Sigma_1}$$

Figure 26: Type rules for unification.

Figure 26 defines the rules for unifying types.

Unify-Union picks a case of the union non deterministically. In our implementation it tries the cases in a deterministic order and picks the first one that succeeds.

4 Semantac

Now that we have an overview of the types of error-detection we want to perform, we can create a more concrete architecture of the tooling proposed in subsection 1.3.

4.1 Implementation

We know that the tooling will need to parse some form of semantics specification that allow for analyses like the ones described in section 2 and 3. Thus the initial step of the tool must be parsing a specification for use in the other tools.

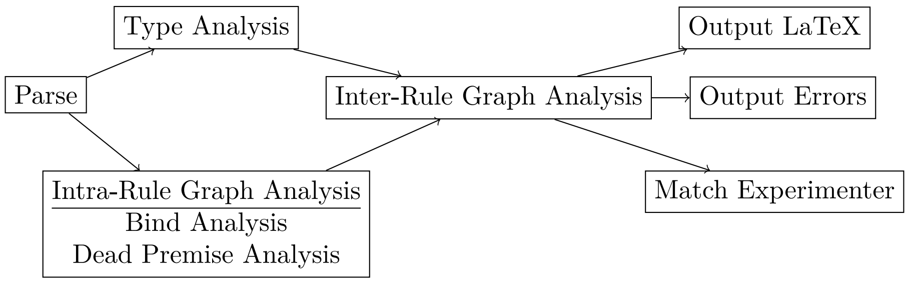


Figure 27: Graph showing the data-flow between and ordering of the stages of Semantac.

Every analysis mentioned uses the specification, however the inter-rule graph analysis need the types given to perform matching on the different configurations. Thus it also depends on the output of the type analysis and must be run after it. The type-check and intra-rule error-detection are shown as bein parrallel processes since the algorithm for checking binds via intra-rule graphs does not use the types of the variables, and the type-checking relies on unification that do not care about premise ordering. After the last analysis is done, Semantaccan either output the errors found in the analysis, output a LaTeXrepresentation of the specification or allow the user to interactively test which rules matches certain inputs.

4.2 Detecting Errors

To evaluate on the usefulness implementation of the semantics tool described in subsection 4.1, we can look at how it handles the errors mentioned in the introduction (subsection 1.1).

```
category Int = "Integer"
category Id = "Identifier"
category Stmt = Syntax
category S in Stmt

category Env = Int -> Id
category env in Env

system (Syntax x Env) => (Syntax x Env)

rule Seq
<S_1, env, sto> => <S_1', env', sto'>
       evn' := evn[a |-> 1]
-------------------------------------
<S_1 ";" S_2, env> => <S_1' ";" S_2>
```

Figure 28: The Seq rule from Figure 1 written in the Semantacsyntax.

The semantics specification in Figure 1 is written in the syntax parseable by Semantac. It contains definitions for the syntactical categories **Id**, **Int** and **Stmt**, with the variable S being an element in **Stmt**. Furthermore it defines *env* as being a function going from **Id** to **Int**. Lastly it defines the transition system as being an arrow $\Rightarrow$: (**Syntax** $\times$ **Env**, **Syntax** $\times$ **Env**).

4.3 Incorrect Configurations

One of the first errors we looked at, was the case where the configurations defined in a rule, did not match up with the definition of the transition system the configuration is being used in. This was the main error that the type-checker is used to find, and is visible both in the missing environment from the conclusion of SEQ where the environment is missing, and in the transition premise of the rule where *sto* is inserted.

```
Type Error: Mismatch between configuration and defined
transition system.
The type of the configuration at example.sem:10:1 to
10:16 does not match the type given in the definition
of the transition system: =>

  The type of the configuration:
    Stmt x Env x #a

  10 | <S_1, env, sto> => <S_1', env', sto'>
     | ^^^^^^^^^^^^^^^^

  The system specifies that it should be:
    Syntax x Env

   7 | system (Syntax x Env) => (Syntax x Env...
     |        ^^^^^^^^^^^^^^

  These two types should match, but they do not.
  Reason: The tuple-types are not the same length

  in Configuration    [example.sem:10:1 to 10:16]
  in Premise          [example.sem:10:1 to 10:38]
  in Rule "Seq"       [example.sem:9:1 to 13:37]
```

Figure 29

The error in Figure 29 shows the type-system displaying an error about the mismatch between the configuration form used in the transition premise and what the transition system defines the initial configuration form to be. The error shows the user both what the types that mismatch are, where in the code the types are defined and a reason why the types do no match. This hopefully is enough to show the tools users which changes are needed in the semantic specification for the configurations and transitions to make sense.

4.4 Unused or Undefined Symbols

Some of the other very important errors for rules in the specification to make sense is the use of variables. As mentioned earlier, the logic of the rules are based on the ordering of premises through the use of variables, so having problems with unused or unbound variables can change the ordering of premises and thus the logic of the rules.

```
Bind Error: Unbound variables in a premise of the rule.
The unbound variables are: sto

 12 | <S_1, env, sto> => <S_1', env', sto'>
    | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

  Variables need to be defined in the initial
  configuration of the rule, in the conclusion
  of a transition premise or by a definition
  premise such as 'sto := 2'

  in Rule "Seq"    [example.sem:11:1 to 15:37]
```

Figure 30

The error displayed in Figure 30 tells the user that one or more variables in a premise is not bound, and where the variable should be in order for it to be defined. The error of unused variables are nearly identical with the explaining message being about the variables not being used in other premises and that is should be marked as unused, so that the tool can insure that the users intent is to not use it.

4.5 Unreachable Premises

Problematic bindings and typos can lead to premises that are essentially "dead code" that will never be reached. This is likely never the users intent, because if it is a legacy premise it is expected that the user comments out the premise, since leaving a premise gives the impression that the premise is still important.

```
Bind Error: a premise of the rule is not reachable via
any variables.
None of the variables in the premise are used in rules
required to reach the conclusion or reachable from the
initial configuration.

 11 |       evn' := evn[a |-> 1]
    |       ^^^^^^^^^^^^^^^^^^^^^

in Rule "Seq"    [example.sem:9:1 to 14:41]
```

Figure 31

The error in Figure 31 is relatively small and simply informs the user that no variables are being used outside the premise in a way that is crucial to the logic of going from the rules initial configuration to the rules final configuration.

5 Discussion

The type checker uses the unification algorithm, which to some extent has the drawback of making type errors harder to explain because of the prevalence of type variables in type matching, the possibility in some type systems of infinite types and the difficulty in algorithmically knowing which type is the correct one in the users eyes. This could possibly have been solved by using the results of the intra-rule graph analysis which gives a valid ordering of the different uses of a variable when there is one. The type Given this information, we could run the inference algorithm on the premises in the given order, which might allow us to simplify the type inference algorithm. This could perhaps lead to a simpler design, but since the type checker was already working by the time where binding analysis was implemented, we did not pursue this. However, having the type checker work independent of binding analysis does have the advantage of allowing us to report type errors while there are bindings errors. This extra information could perhaps be helpful in diagnosing the problem with the specification.

Some aspects of the type rules specification can perhaps be simplified - in particular the specification might perform more substitutions than necessary. This is an artifact of us having derived a pure mathematical specification from an imperative-style implementation which mutates the substitutions and bindings, and which carries a context.

6 Conclusion

Treating Structural Operational Semantics as a programming language is an approach that allows various techniques to be applied to specification to detect errors which are common in research around programming languages. There are many known tools and techniques not tried in this paper, which could possible allow greater results for detecting errors in semantics, and the approach of treating semantics as a language might allow more flexibility in using over- or under-approximative results to help a language designer. It might be interesting using this tool as a base for researching if having a compiler-like tool available actually help people initially getting into workng with and creating SOS specifications.

Bibliography

Felleisen, Matthias, and Robert Hieb. 1992. “The Revised Report on the Syntactic Theories of Sequential Control and State.” *Theor. Comput. Sci.* 103 (2): 235–71. https://doi.org/10.1016/0304-3975(92)90014-7.

Hartel, Pieter H. 1999. “LETOS - a Lightweight Execution Tool for Operational Semantics.” *Softw. Pract. Exp.* 29 (15): 1379–1416. https://doi.org/10.1002/(SICI)1097-024X(19991225)29:15/%3C1379::AID-SPE286/%3E3.0.CO;2-V.

Hüttel, Hans. 2010. *Transitions and Trees - an Introduction to Structural Operational Semantics.* Cambridge University Press. http://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/transitions-and-trees-introduction-structural-operational-semantics.

Klein, Casey, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. 2012. “Run Your Research: On the Effectiveness of Lightweight Mechanization.” In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, edited by John Field and Michael Hicks, 285–96. ACM. https://doi.org/10.1145/2103656.2103691.

Plotkin, Gordon D. 2004. “The Origins of Structural Operational Semantics.” *J. Log. Algebraic Methods Program.* 60-61: 3–15. https://doi.org/10.1016/j.jlap.2004.03.009.