Shoggoth: A Formal Foundation for Strategic Rewriting

XUEYING QIN, University of Edinburgh, UK
LIAM O'CONNOR, University of Edinburgh, UK
ROB VAN GLABBEEK, University of Edinburgh, UK and UNSW, Australia
PETER HÖFNER, Australian National University, Australia
OHAD KAMMAR, University of Edinburgh, UK
MICHEL STEUWER, Technische Universität Berlin, Germany and University of Edinburgh, UK

Rewriting is a versatile and powerful technique used in many domains. *Strategic rewriting* allows programmers to control the application of rewrite rules by composing individual rewrite rules into complex rewrite strategies. These strategies are semantically complex, as they may be nondeterministic, they may raise errors that trigger backtracking, and they may not terminate.

Given such semantic complexity, it is necessary to establish a formal understanding of rewrite strategies and to enable reasoning about them in order to answer questions like: How do we know that a rewrite strategy terminates? How do we know that a rewrite strategy does not fail because we compose two incompatible rewrites? How do we know that a desired property holds after applying a rewrite strategy?

In this paper, we introduce Shoggoth: a formal foundation for understanding, analysing and reasoning about strategic rewriting that is capable of answering these questions. We provide a denotational semantics of System S, a core language for strategic rewriting, and prove its equivalence to our big-step operational semantics, which extends existing work by explicitly accounting for divergence. We further define a *location-based weakest precondition calculus* to enable formal reasoning about rewriting strategies, and we prove this calculus sound with respect to the denotational semantics. We show how this calculus can be used in practice to reason about properties of rewriting strategies, including termination, that they are well-composed, and that desired postconditions hold. The semantics and calculus are formalised in Isabelle/HOL and all proofs are mechanised.

CCS Concepts: • **Theory of computation** → *Denotational semantics*; *Operational semantics*; *Axiomatic semantics*; Hoare logic; **Pre- and post-conditions**; **Rewrite systems**.

Additional Key Words and Phrases: strategic rewriting, program transformation, weakest preconditions, semantics, mechanised formalisation

ACM Reference Format:
Xueying Qin, Liam O'Connor, Rob van Glabbeek, Peter Höfner, Ohad Kammar, and Michel Steuwer. 2024. Shoggoth: A Formal Foundation for Strategic Rewriting. *Proc. ACM Program. Lang.* 8, POPL, Article 3 (January 2024), 29 pages. https://doi.org/10.1145/3633211

1 INTRODUCTION

Strategic rewriting allows programmers to compose rewrite rules and control their application. Dedicated strategy languages, such as Stratego [Visser 2001; Visser et al. 1998] and more recently

Authors' addresses: Xueying Qin, University of Edinburgh, Edinburgh, UK, xueying.qin@ed.ac.uk; Liam O'Connor, University of Edinburgh, Edinburgh, UK, l.oconnor@ed.ac.uk; Rob van Glabbeek, University of Edinburgh, Edinburgh, UK and UNSW, Sydney, Australia, rvg@cs.stanford.edu; Peter Höfner, Australian National University, Canberra, Australia, peter.hoefner@anu.edu.au; Ohad Kammar, University of Edinburgh, Edinburgh, UK, ohad.kammar@ed.ac.uk; Michel Steuwer, Technische Universität Berlin, Berlin, Germany and University of Edinburgh, Edinburgh, UK, michel.steuwer@tu-berlin.de.

ACM 2475-1421/2024/1-ART3
https://doi.org/10.1145/3633211

Elevate [Hagedorn et al. 2020, 2023], provide combinators for composing rewrite rules into larger strategies, as well as traversals to describe the location at which rewrite strategies are applied.

Strategic rewriting has important practical applications. Stratego is used to specify the semantics of programming languages by writing interpreters with rewrite strategies in the Spoofax language workbench [Wachsmuth et al. 2014]. Elevate is used to describe compiler optimisations for generating fast code achieving competitive performance to the state-of-the art machine learning compiler TVM [Hagedorn et al. 2020]. Strategic rewriting is also used in domains ranging from generic programming [Lämmel and Visser 2002] to tactic languages in proof assistants [Sozeau 2014].

Compositions of rewrites easily become complex. For example, Hagedorn et al. [2020] report that for performing their compiler optimisations up to 60, 000 rewrite steps are required. To orchestrate such long rewrite sequences, strategy languages provide various combinators for composing strategies together and traversals for applying strategies to different sub-expressions within the given abstract syntax tree. Together with support for recursion, these combinators and traversals are capable of modelling the complex rewrite sequences required in practical applications.

This capability comes at the cost of semantic complexity, as strategies can be nondeterministic, they may give an error which triggers backtracking, and they may diverge due to the presence of general recursion. Such a combination of features introduces a lot of semantic subtleties, which make it easy to define not well-behaved strategies by mistake. For example, a strategy that does not terminate as it repeatedly tries to apply a rewrite. Similarly, it is easy to compose incompatible rewrites that will fail for every possible input expression. Finally, even if a rewrite strategy successfully terminates, it may not do what it was supposed to do by rewriting the input expression into an undesired form.

The goal of this paper is to provide a rigorous treatment of strategic rewriting, that we believe lacks so far. Considering that strategic rewriting has various application domains but has problematic behaviours, a rigorous formal understanding of strategic rewriting is required to model and analyse its semantic subtleties as well as reason about the execution of strategies. Therefore, we present Shoggoth: a formal foundation for reasoning about strategic rewriting.

We start with introducing the formal syntax of *System S*, a formal core strategy language originally introduced by Visser and Benaissa [1998]. Some example strategies are sketched to give the gist of strategic rewriting as well. We then give a comprehensive semantic accounting of strategic rewriting languages. We define a *denotational semantics* for System S, which originally had been given a big-step operational semantics. Our denotational semantics accounts for non-determinism and errors, and, unlike previous work, also explicitly models divergence. We formalise an extended big-step operational semantics which accounts for diverging executions, and formally prove the equivalence of our two models via soundness and computational adequacy theorems. All of our results have been mechanised in Isabelle/HOL [Nipkow et al. 2002].

To facilitate formal reasoning about rewriting strategies, we define a *weakest precondition calculus* that for a given postcondition computes the weakest precondition that must hold in order for the given strategy to execute successfully and satisfy the postcondition. Because traversals allow us to apply strategies to sub-expressions of the input expression, we must know not just which rewrite rules are to be applied, but also *where* in the input expression they are to be applied, in order to determine the weakest precondition. To accomplish this, our weakest precondition calculus is *location-based*: weakest preconditions are not just based on the given strategy and desired postcondition, but also depend on the location at which the strategy is to be applied in the input expression. We have mechanised the definition of the weakest precondition calculus in Isabelle/HOL and formally proven its soundness with respect to the denotational semantics. [1]

[1] Our mechanisation can be found at the Archive of Formal Proofs.

Finally, we show how to use the weakest precondition calculus to reason about rewrite strategies by applying it to various examples, including termination, that a strategy is well-composed, and that a rewrite strategy satisfies a particular postcondition after its execution. One of our examples is a strategy for $\beta\eta$-normalisation taken from the Elevate project by Hagedorn et al. [2020], demonstrating the applicability of our work to practical scenarios.

In summary, we make the following contributions:

- We design, formalise and mechanise using Isabelle/HOL the semantics of System S, including both denotational and operational models with a full accounting of nondeterminism, errors, and divergence. We prove these two semantics equivalent (Section 3).
- We design, formalise and mechanise using Isabelle/HOL a location-based weakest precondition calculus for System S. We prove its soundness with respect to the denotational semantics (Section 4).
- We demonstrate how to use the weakest precondition calculus to prove practical useful properties of strategic rewriting (Section 5):
  - that a strategy terminates, i.e., that is does not diverge;
  - that a strategy is well-composed, i.e., that there exist input expressions for which the strategy execution will succeed;
  - that a desired property is satisfied after execution of the strategy.

Before stepping into the formalisation of System S, in the next section we present the syntax of System S as well as some example strategies to facilitate the understanding of strategic rewriting.

2 THE SYNTAX OF SYSTEM S

System S [Visser and Benaissa 1998] is a core calculus providing basic constructs of strategic rewriting, including atomic strategies (rewrite rules) and operators composing strategies and performing expression traversals in an abstract syntax tree (AST). A successful execution of a strategy transforms an expression into some other expression while preserving its semantics. The expressions being rewritten can either be *Leaf*s or *n*odes, in general, taking the form of:

$$\text{Expression}(\mathbb{E}) \quad e ::= \quad \mathit{Leaf} \;\mid\; \overset{n}{\underset{e\ \ e}{\wedge}}$$

Figure 1 presents the syntax of strategies in System S. We use $\mathbb{S}$ to denote the set of all strategies. Variables, atomic strategies, SKIP and ABORT are *basic strategies*. Basic strategies are not decomposable. An atomic strategy is simply a rewrite rule. For instance, the commutativity of addition add_{com} and commutativity of multiplication $mult_{com}$ are atomic strategies:

$$add_{com} : a + b \rightsquigarrow b + a \qquad \text{Commutativity of addition}$$
$$mult_{com} : a * b \rightsquigarrow b * a \qquad \text{Commutativity of multiplication}$$

SKIP can always be executed successfully while executing ABORT would always cause failure. To compose strategies, one can make use of *combinators* including sequential composition (;), left choice (<+) and nondeterministic choice (<+>). Sequential composition instructs to execute two strategies one after the other. Left choice prefers executing the strategy at the left hand side of the combinator over the strategy at the right hand side of the operator while nondeterministic choice decides to execute one of the given two strategy nondeterministically. In addition, *one*, *some* and *all* are *traversals* that navigate within the AST. Intuitively, $one(s)$ applies s to one immediate sub-expression of an input expression, $some(s)$ applies s to as many immediate sub-expressions of

$$\begin{aligned}\text{Strategy}(\mathbb{S}) \quad s ::= {} & \mathit{atomic} \mid X \mid \text{SKIP} \mid \text{ABORT} \\ & \mid s \ ; \ s \mid s <\!\!+ \ s \mid s <\!\!+\!\!> s \\ & \mid \mathit{one}(s) \mid \mathit{some}(s) \mid \mathit{all}(s) \\ & \mid \mu X.s\end{aligned}$$

Fig. 1. The Syntax of System S

an input expression as possible and $\mathit{all}(s)$ applies s to all immediate sub-expressions of an input expression. Lastly, System S provides a *fixed-point operator* to model recursion.

Comparison of the expressiveness to the original System S. One difference between our formalism and the original System S is that we abstract away the term building details for atomic strategies, instead modelling atomic strategies as partial functions. We believe that applying this abstraction does not limit the expressiveness of our system. In fact, the purpose of such design is to allow the flexibility of the term languages, not only limited to the original System S, but also capturing other strategic rewriting languages that use term constructs that are different from System S. Moreover, this design enables us to focus on reasoning about properties of compositions of rewriting strategies that hold independent of the term building behaviour.

Composing strategies. We can compose strategies together with these combinators, traversals and the fixed-point operator to define more strategies. For example, we define a strategy $\mathit{try}(s)$ using left choice and SKIP which attempts to apply a strategy s to an input expression. If an error occurs, then it will leave the input expression unchanged by executing the strategy SKIP:

$$\mathit{try}(s) \coloneqq s <\!\!+ \text{SKIP}$$

With the fixed-point operator and sequential composition, we can then define a strategy $\mathit{repeat}(s)$ which keeps applying a strategy s to an input expression until its no longer applicable:

$$\mathit{repeat}(s) \coloneqq \mu X.\mathit{try}(s \ ; \ X)$$

With the fixed-point operator, the traversal $\mathit{one}(s)$ and left choice, we can define top-down and bottom-up traversals in an AST:

$$\mathit{topDown}(s) \coloneqq \mu X.(s <\!\!+ \mathit{one}(X)) \qquad \mathit{bottomUp}(s) \coloneqq \mu X.(\mathit{one}(X) <\!\!+ s)$$

We can further compose $\mathit{repeat}(s)$ and $\mathit{topDown}(s)$ to define a strategy $\mathit{normalise}(s)$, which keeps applying a strategy s to all sub-expressions of an input expression until it is no longer applicable:

$$\mathit{normalise}(s) \coloneqq \mathit{repeat}(\mathit{topDown}(s))$$

The *normalise* strategy is very commonly used to express program transformations. Given *beta* and *eta* reductions for λ-expressions, we can use the normalisation strategy $\mathit{normalise}(\mathit{beta} <\!\!+ \mathit{eta})$ for normalising an input λ-expression into its $\beta\eta$-normal form.

As previously mentioned, the composition of strategies can be invalid and the executions of strategies are not always successful. For instance, the strategy $\mathit{mult}_{com} \ ; \ \mathit{add}_{com}$ is not well composed since it cannot be successfully executed on any input expression. $\mathit{repeat}(\text{SKIP})$ is a strategy that cannot terminate. Although $\mathit{normalise}(\mathit{beta} <\!\!+ \mathit{eta})$ can certainly be successfully executed on some input expressions, on other inputs it may not terminate. It is important to know that when it terminates, it will indeed leave the expression in $\beta\eta$-normal form.

To reason about the successful and unsuccessful executions of strategies, we design the *location-based weakest precondition calculus* which is discussed in section 4. With this calculus, we are

able to detect *bad* strategies that do not have successful executions, like $mult_{com}$; add_{com} and *repeat*(SKIP), by concluding that there is no input expression that can be successfully rewritten by such strategies into a desired form. Also, for a *good* strategy that has successful executions, we are able to distinguish inputs that indeed lead to successful executions of the strategy and inputs that lead to erroneous or diverging executions. Such reasoning power is demonstrated in section 5.

To design the location-based weakest precondition calculus, we need to understand the behaviours of executing these strategies in System S. Therefore, before introducing the calculus and its reasoning power, we firstly study the formal semantics of System S.

3 THE SEMANTICS OF SYSTEM S

For given collections of expressions $\mathbb{E}$, System S defines nondeterministic execution for given strategies that can result in expressions or errors. We extend the original System S by allowing divergence as a possible result of executing a strategy. Therefore, applying a strategy to an expression can result in expressions, an error or divergence.

3.1 The Plotkin Powerdomain

We provide a denotational semantics of System S as an instance of Plotkin's powerdomain construction [Plotkin 1976], which allows us to assign least fixed points as the semantics of the recursion construct. An *ω-complete partial order* (*ω-cpo*) is a partially ordered set $(X, \preceq)$ in which each ω-chain $(x_1 \preceq x_2 \preceq x_3 \preceq \dots)$ has a least upper bound. A function $f : X \to X$ on such a set is *continuous* if for each ω-chain $x_1 \preceq x_2 \preceq x_3 \preceq \dots$ with least upper bound x, one has that $f(x)$ is the least upper bound of the set $\{f(x_1), f(x_2), f(x_3), \dots\}$. A continuous function is certainly *monotone*, in the sense that $x_1 \preceq x_2$ implies $f(x_1) \preceq f(x_2)$ – this follows by considering the ω-chain $x_1 \preceq x_2 \preceq x_2 \preceq x_2 \preceq \dots$, and its least upper bound x_2. Now Kleene's fixed-point theorem says that each continuous function f on an ω-cpo with a least element has a least fixed point.

Consider a nondeterministic, possibly diverging, algorithm that transforms values into values. If $\mathcal{V}$ is the set of values, this algorithm can be modelled as a function $f : \mathcal{V} \to \mathscr{P}_{\neg\emptyset}(\mathcal{V}_\perp)$, where $\mathscr{P}_{\neg\emptyset}(X)$ is the set of non-empty subsets of X, the non-empty-powerset, and $\mathcal{V}_\perp := \mathcal{V} \uplus \{\perp\}$ is the set in which we embed $\mathcal{V}$ together with a new element $\perp$. The newly added element $\perp$ represents the outcome where the algorithm diverges. We equip the set $\mathcal{V}_\perp$ with a partial order by defining:

$$x \preceq y \iff x = \perp \vee x = y \,.$$

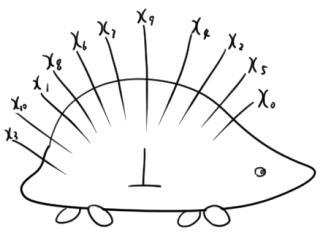
This fits with the intuition that $\perp$ represents a computation that has not yet terminated, and $x \preceq y$ holds when y is a later stage of the computation x.

Terminated computations are identified by the values they compute. We compare sets of values using the Egli-Milner ordering:

$$A \preceq B \iff (\forall x \in A.\ \exists y \in B.\ x \preceq y) \wedge (\forall y \in B.\ \exists x \in A.\ x \preceq y)$$

Lifting a partial order from elements to sets in this fashion always yields a preorder. For a *flat domain* $\mathcal{V}_\perp$, $\preceq$ is a partial order on $\mathscr{P}_{\neg\emptyset}(\mathcal{V}_\perp)$. It is characterised by:

$$A \preceq B \iff A = B \vee ((\perp \in A) \wedge A \backslash \{\perp\} \subseteq B)$$



(Porcupine ordering)

The resulting poset $\mathscr{P}_{\neg\emptyset}(\mathcal{V}_\perp)$ is an ω-cpo. Each ω-chain either enters a spine of the porcupine, and thus contains a largest element which is its least upper bound, or $\perp$ is a member of all elements in the chain, so that its least upper bound is simply the union of all sets in the chain.

Aside on the powerdomain construction and the Egli-Milner ordering. To give some further insight into the powerdomain construction and the Egli-Milner ordering, recall the following well-known characterisation. Hennessy and Plotkin [1979, Remark after Lemma 3.5] show that Plotkin's [1976] powerdomain construction extends to all ω-complete partial orders (ω-cpos) by sending each ω-cpo to the *free* semi-lattice over it. In detail, given an ω-cpo X, we define a *free semi-lattice over* X as an ω-cpo DX, together with a Scott-continuous function $\eta : X \to DX$ and a Scott-continuous binary operation: $\vee : (DX)^2 \to DX$ that is associative, commutative, and idempotent. A free semi-lattice always exists, but its explicit description may be complicated. Hennessy and Plotkin show that, when ω-cpo is ω-algebraic, we can construct the free semi-lattice explicitly by taking $DX := \mathscr{P}_{\neg\emptyset}X$ to be the powerdomain construction with the Egli-Milner ordering, $\eta(x) := \{x\}$ as the embedding of X into this semi-lattice, and sub-set union as the binary operation. So in a specific and technical sense, the powerdomain DX is the simplest extension of the ω-cpo X with an associative, idempotent and commutative binary operator. *(end of aside)*

In our mechanised Isabelle/HOL formalisation, we opt to use posets that are complete with respect to all chains, not merely countable or directed ones, without maintaining continuity as an assumption. The stronger assumption on posets allows us to weaken the assumption on functions: we only require monotonicity to ensure existence of fixed points. This choice was made purely for ease of formalisation, as Isabelle/HOL already includes a library for chain-complete partial orders. While this means that our domain may contain monotone functions that do not correspond to any expressible strategy, and that Hennessy and Plotkin's characterisation does not directly apply, our meta-theoretic results below show how to relate our semantics to the operational semantics, and our reasoning examples show that this semantics suffices to reason about practically interesting examples. We conjecture that our results will easily carry over to a semantics defined with ω-cpos.

3.2 Formalised Denotational Semantics

We now present and discuss the denotational semantics for System S, capturing successful and erroneous executions of strategies as well as nondeterminism, divergence and recursion. A strategy is a nondeterministic algorithm/function that rewrites expressions into expressions. This nondeterministic algorithm can sometimes yield an error *err* instead of an expression, and it might fail to terminate. In the latter case, we say that it yields the value *div*. Formally, we instantiate Plotkin's powerdomain construction from the previous section by setting $\mathcal{V} := \mathbb{E} \cup \{err\}$ and $\bot := div$, noting it is a flat domain. We denote the resulting powerdomain by:

$$\mathfrak{D}_p := \mathscr{P}_{\neg\emptyset}(\mathbb{E} \cup \{err\} \cup \{div\}), \quad \text{ordered by} \quad A \preceq B \iff A = B \vee ((div \in A) \wedge A \backslash \{div\} \subseteq B).$$

We define the denotational semantics of System S over the point-wise lifting of the powerdomain:

$$\mathfrak{D} = \mathbb{E} \to \mathfrak{D}_p$$

To define the denotational semantics of strategies in a concise manner, we provide semantic combinators and traversals that encapsulate the meaning of syntactic combinators and traversals.

Figure 2 illustrates the definitions of the combinators. The definition of sequential composition $s ;_s t$ is straightforward, indicating that the execution of the strategy t depends on the result of applying s to the input expression e. If applying s to e results in an error or divergence, the sequential composition will produce an error and divergence, respectively. Otherwise, the result of the sequential composition $s ;_s t$ is produced by applying t to the expression obtained by the execution of s. The definition of left choice $s <+_s t$ prioritises the execution of the strategy s over t. The strategy t will only be executed if the execution of s produces an error. Our treatment of nondeterminism is *demonic* with respect to divergence while *angelic* with respect to errors. If either the execution of s or t divergences, then the nondeterministic choice $s <+>_s t$ diverges as well. The

$$(\ ;_s\ ) : \mathfrak{D} \to \mathfrak{D} \to \mathfrak{D}$$
$$(s\ ;_s\ t)(e) = \bigcup \{t(e') \mid e' \in s(e) \cap \mathbb{E}\} \cup \{r \mid r \in s(e) \cap \{div, err\}\}$$
(Sequential composition)
$$(<+_s) : \mathfrak{D} \to \mathfrak{D} \to \mathfrak{D}$$
$$(s <+_s t)(e) = (s(e) \setminus \{err\}) \cup \{e' \mid e' \in t(e) \wedge err \in s(e)\}$$
(Left choice)
$$(<+>_s) : \mathfrak{D} \to \mathfrak{D} \to \mathfrak{D}$$
$$(s <+>_s t)(e) = \{e' \mid e' \in s(e) \cap \mathbb{E}\} \cup \{div \mid div \in s(e)\} \cup \{e' \mid e' \in t(e) \cap \mathbb{E}\} \cup \{div \mid div \in t(e)\} \cup \{err \mid err \in s(e) \cap t(e)\}$$
(Nondeterministic choice)

Fig. 2. Semantic Combinators of System S

nondeterministic choice will only result in an error if both executions of s and t result in an error. When both s and t give cause for a successful execution, the choice is nondeterministic.

These combinators are sufficient for composing strategies applied to the root of an AST. System S also provide traversals *one*, *some* and *all* to apply strategies to sub-expressions. Their semantics are shown in figure 3. The traversal $one_s(s)(e)$ nondeterministically chooses one immediate sub-expression of e and applies strategy s to it. The treatment of nondeterminism here is again demonic with respect to divergence and angelic with respect to errors. If applying s to one of the sub-expressions results in divergence, $one_s(s)$ will diverge. An error will only occur when e has no sub-expression or applying s to all sub-expressions of e results in error. The traversal $some_s(s)(e)$ applies s to as many immediate sub-expressions of e as possible. Its divergence and erroneous situations are the same as one_s. The successful execution of $all_s(s)$ on an input expression e requires successful application of s to all immediate sub-expressions of e or e being a *Leaf*. If applying s to one sub-expression leads to an error or divergence, $all_s(s)(e)$ yields *err* or *div*, respectively. For simplicity of the presentation and illustration, we have restricted ourselves to binary trees in this paper. However, the traversals can easily be generalised to ASTs with wider branching.

With the semantic combinators and semantic traversals introduced, we provide the denotational semantics for System S shown in figure 4. The semantics of a strategy is modelled as a function that takes in a *semantic environment* ξ, which is a function mapping variables to elements of $\mathfrak{D}$.

The semantics of a variable consists of looking up the variable in a given semantic environment. We model an atomic strategy as a partial function, which can successfully rewrite an input expression into an output expression when it is defined for the input expression. When an atomic strategy is not defined for an input expression, applying it to the input expression will result in an error. SKIP is a strategy that always rewrites an input expression to itself while ABORT is a strategy that always produces an *err*. The denotational semantics of combinators and traversals are straightforwardly defined with the semantic combinators and traversals. Lastly, the semantics of the fixed-point operator is the least fixed point in our domain, where we extend the semantic environment with a mapping from the syntactic fixed-point variable to the fixed point in our domain. We denote this environment extension with the syntax $\xi[X \mapsto d]$.

The denotational semantics is monotone. Given two environments ξ_1 and ξ_2, if the values obtained from looking up the variables in the environments satisfy the ordering $\xi_1(X) \preceq \xi_2(X)$ for any

$(one_s) : \mathfrak{D} \to \mathfrak{D}$

$$one_s(s)(e) = \{ \overset{n}{\underset{e_1' \quad e_2}{\wedge}} \mid e = \overset{n}{\underset{e_1 \quad e_2}{\wedge}} \wedge e_1' \in s(e_1) \cap \mathbb{E} \} \cup \{ \overset{n}{\underset{e_1 \quad e_2'}{\wedge}} \mid e = \overset{n}{\underset{e_1 \quad e_2}{\wedge}} \wedge e_2' \in s(e_2) \cap \mathbb{E} \}$$
$$\cup \{ div \mid e = \overset{n}{\underset{e_1 \quad e_2}{\wedge}} \wedge div \in s(e_1) \cup s(e_2) \}$$
$$\cup \{ err \mid e = Leaf \vee (e = \overset{n}{\underset{e_1 \quad e_2}{\wedge}} \wedge err \in s(e_1) \cap s(e_2)) \}$$

(One)

$(some_s) : \mathfrak{D} \to \mathfrak{D}$

$$some_s(s)(e) = \{ \overset{n}{\underset{e_1' \quad e_2'}{\wedge}} \mid e = \overset{n}{\underset{e_1 \quad e_2}{\wedge}} \wedge e_1' \in s(e_1) \cap \mathbb{E} \wedge e_2' \in s(e_2) \cap \mathbb{E} \}$$
$$\cup \{ \overset{n}{\underset{e_1' \quad e_2}{\wedge}} \mid e = \overset{n}{\underset{e_1 \quad e_2}{\wedge}} \wedge e_1' \in s(e_1) \cap \mathbb{E} \wedge err \in s(e_2) \}$$
$$\cup \{ \overset{n}{\underset{e_1 \quad e_2'}{\wedge}} \mid e = \overset{n}{\underset{e_1 \quad e_2}{\wedge}} \wedge e_2 \in s(e_2) \cap \mathbb{E} \wedge err \in s(e_1) \}$$
$$\cup \{ div \mid e = \overset{n}{\underset{e_1 \quad e_2}{\wedge}} \wedge div \in s(e_1) \cup s(e_2) \}$$
$$\cup \{ err \mid e = Leaf \vee (e = \overset{n}{\underset{e_1 \quad e_2}{\wedge}} \wedge err \in s(e_1) \cap s(e_2)) \}$$

(Some)

$(all_s) : \mathfrak{D} \to \mathfrak{D}$

$$all_s(s)(e) = \{ Leaf \mid e = Leaf \} \cup \{ \overset{n}{\underset{e_1' \quad e_2'}{\wedge}} \mid e = \overset{n}{\underset{e_1 \quad e_2}{\wedge}} \wedge e_1' \in s(e_1) \cap \mathbb{E} \wedge e_2' \in s(e_2) \cap \mathbb{E} \}$$
$$\cup \{ div \mid e = \overset{n}{\underset{e_1 \quad e_2}{\wedge}} \wedge div \in s(e_1) \cup s(e_2) \}$$
$$\cup \{ err \mid e = \overset{n}{\underset{e_1 \quad e_2}{\wedge}} \wedge err \in s(e_1) \cup s(e_2) \}$$

(All)

Fig. 3. Semantic Traversals of System S

variable X, the values obtained from evaluation of a strategy s with these environments should also satisfy the ordering $[\![s]\!]\xi_1 \preceq [\![s]\!]\xi_2$. Formally, we present the monotonicity theorem 3.1:

Theorem 3.1 (Semantics monotonicity theorem). *For given environments ξ_1 and ξ_2, and strategy s we have:*

$$\frac{\forall X. \xi_1(X) \preceq \xi_2(X)}{[\![s]\!]\xi_1 \preceq [\![s]\!]\xi_2}$$

$$\begin{aligned}
\text{Variable}(\mathbb{V}) &\quad X\ Y\ Z \ldots \\
\text{Semantic Environment}(\Gamma_S) &\quad \xi : \mathbb{V} \to \mathfrak{D}
\end{aligned}$$

$$[\![\ \mathbb{S}\ ]\!] : \Gamma_S \to \mathfrak{D}$$

$$[\![X]\!]\xi = \xi X$$

$$[\![atomic]\!]\xi = \lambda e.\{atomic(e) \mid atomic(e)\ \mathbf{def}\} \cup \{err \mid atomic(e)\ \mathbf{undef}\}$$

$$[\![\text{SKIP}]\!]\xi = \lambda e.\{e\}$$

$$[\![\text{ABORT}]\!]\xi = \lambda e.\{err\}$$

$$[\![s\ ;\ t]\!]\xi = [\![s]\!]\xi\ ;_s\ [\![t]\!]\xi \qquad \text{(Sequential composition)}$$

$$[\![s\ <+\ t]\!]\xi = [\![s]\!]\xi\ <+_s\ [\![t]\!]\xi \qquad \text{(Left choice)}$$

$$[\![s\ <+>\ t]\!]\xi = [\![s]\!]\xi\ <+>_s\ [\![t]\!]\xi \qquad \text{(Nondeterministic choice)}$$

$$[\![one(s)]\!]\xi = one_s([\![s]\!]\xi) \qquad \text{(One)}$$

$$[\![some(s)]\!]\xi = some_s([\![s]\!]\xi) \qquad \text{(Some)}$$

$$[\![all(s)]\!]\xi = all_s([\![s]\!]\xi) \qquad \text{(All)}$$

$$[\![\mu X.s]\!]\xi = \mu\mathcal{X}.[\![s]\!](\xi[X \mapsto \mathcal{X}]) \qquad \text{(Fixed point)}$$

Fig. 4. Denotational Semantics of System S

We prove this theorem in Isabelle/HOL by structural induction on the strategy s.

3.3 Formalised Big-Step Operational Semantics

In this section, we present the formalised big-step operational semantics of System S, with our extension allowing for divergent strategies. Figure 5 depicts the big-step operational semantics for the non-diverging cases of System S. These cases are essentially the same as those of Visser and Benaissa [1998], albeit with the aforementioned simplification to binary trees applied.[2] The semantic rules are given in a straightforward way.

On top of these rules for terminating cases, we define the semantics for divergence as the *coinductive* judgement [Leroy and Grall 2009] satisfying the rules shown in figure 6. Here we use $e \xrightarrow[\infty]{s}$ to indicate that the evaluation of an expression e by a strategy s leads to divergence.

3.4 The Denotational Semantics is Equivalent to The Big-Step Operational Semantics

In section 3.2 and section 3.3, we have provided two styles of semantics for System S. It is essential to prove that these two semantics are equivalent, since we would like our formal semantics to provide unambiguous and unique interpretation of strategies in System S. In addition, with the equivalence of these two semantics established, we only need to refer to one of them to prove some properties of System S and they should also hold for the other semantics.

We reason about the equivalence between the denotational semantics and big-step operational semantics via computational soundness and computational adequacy theorems. More specifically, we have a pair of computational soundness and adequacy theorems to relate the non-diverging cases and a pair of computational soundness and adequacy theorems to relate the diverging cases.

[2] Visser and Benaissa [1998] denote error by $\uparrow$.

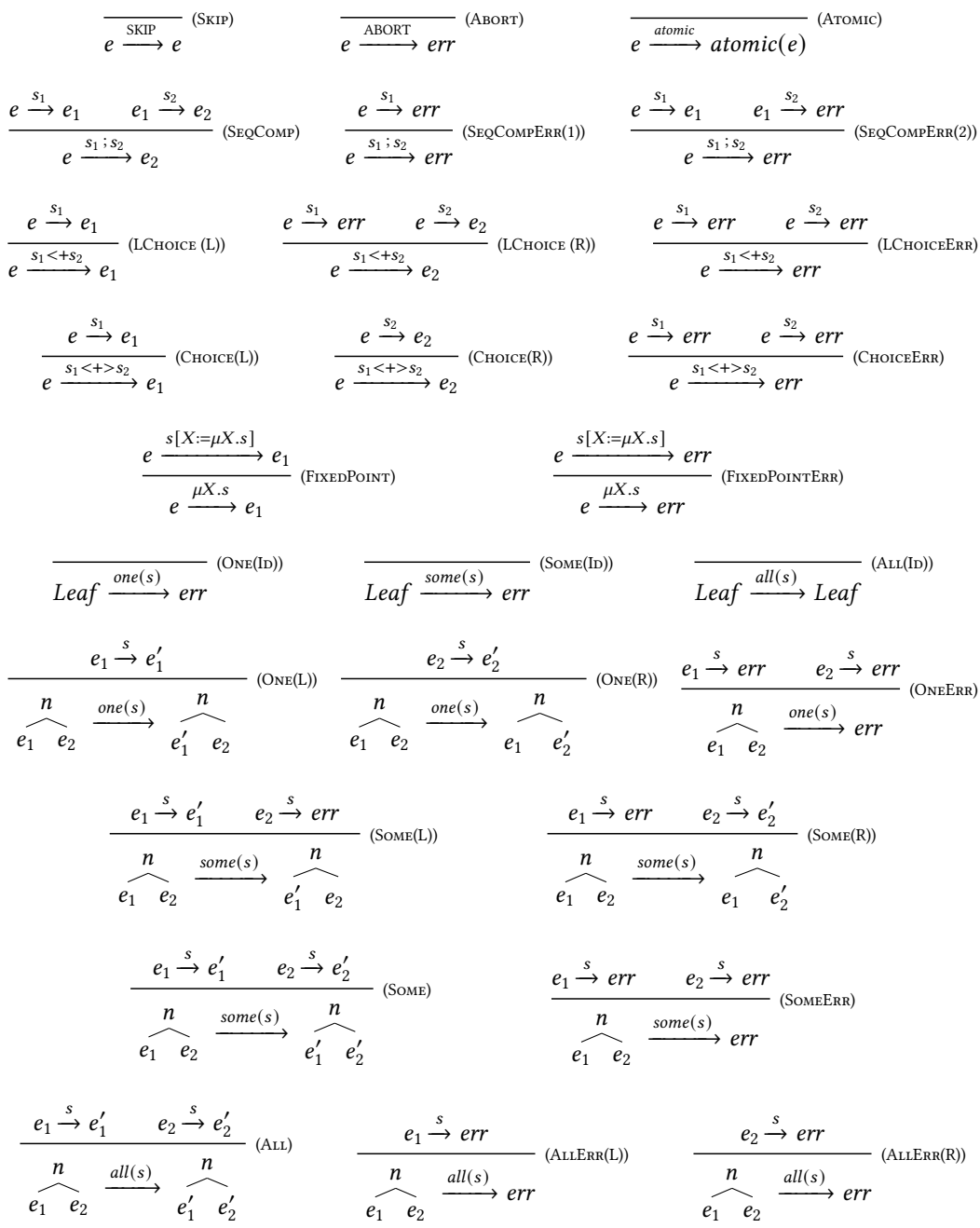


Fig. 5. Big-step operational semantics of non-diverging cases

Firstly, we show that if an expression e is evaluated to another expression or an error using the big-step operational semantics of a strategy $s_{\clubsuit}$, this result must also be in the set obtained by executing the denotational semantics of $s_{\clubsuit}$ with the given expression e. Formally, this is described

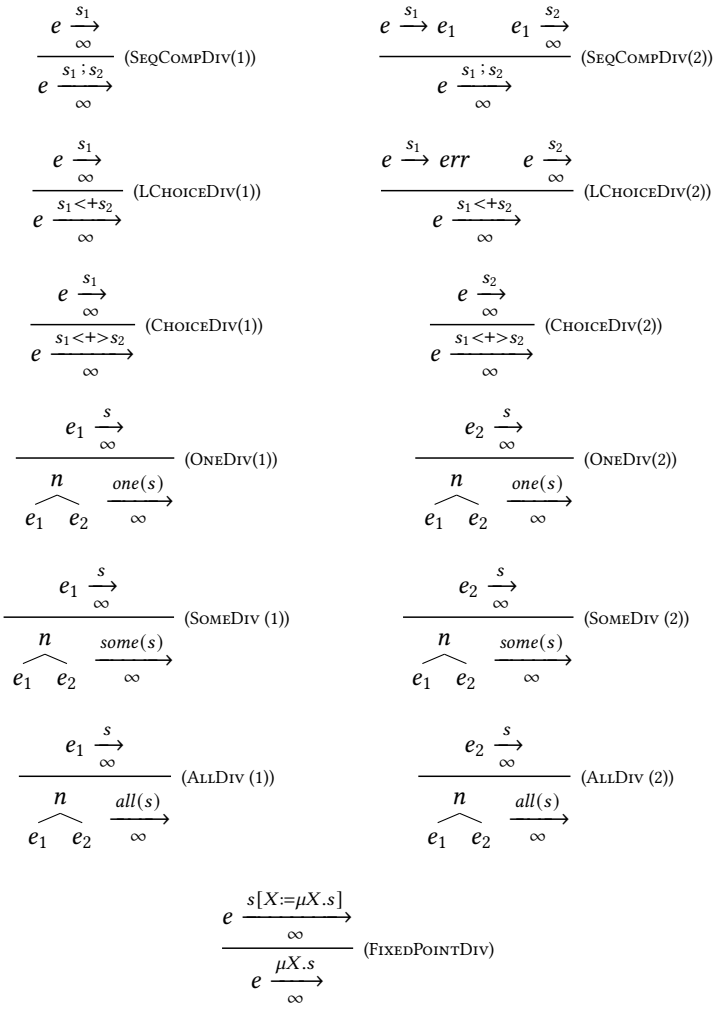


Fig. 6. Big-step operational semantics of diverging cases

by our first computational soundness theorem 3.2. The subscript ♣ indicates that $s_\clubsuit$ is a *closed* strategy: a strategy with no free variables, i.e. $fv(s_\clubsuit) = \emptyset$.

Theorem 3.2 (Computational soundness theorem one). *For a given closed strategy $s_\clubsuit$, and any environment ξ, we have for an arbitrary expression e and result r:*

$$\frac{e \xrightarrow{s_\clubsuit} r}{r \in [\![s_\clubsuit]\!] \xi e}$$

We prove this by induction on the derivation of $e \xrightarrow{s_\clubsuit} r$ from the rules of Figure 5. As the strategy $s_\clubsuit$ is always closed, to instantiate our inductive hypothesis in the cases for the fixed-point operator, we make use of the following substitution lemma 3.3 to semantically relate the syntactic substitution of a closed strategy $s_\clubsuit$ for a variable X in s with the strategy s under the environment where X maps to the semantics of $s_\clubsuit$.

Lemma 3.3 (Substitution lemma one).

$$[\![s[X := s_\clubsuit]]\!]\xi = [\![s]\!]\xi[X \mapsto ([\![s_\clubsuit]\!]\xi)]$$

This lemma can easily be generalised to allow $s_\clubsuit$ to instead be an open strategy, so long as X is not free in $s_\clubsuit$, however our operational semantics only ever substitutes closed strategies, thus this generalisation is not necessary to prove our semantic equivalence theorems.

We now prove a computational adequacy theorem, the converse of the computational soundness theorem 3.2. It states that if a non-diverging result r is one of the results of the denotational semantics for a closed strategy $s_\clubsuit$ with an input expression e, then the big-step operational semantics of $s_\clubsuit$ with the input expression e will produce the same result.

Theorem 3.4 (Computational adequacy theorem one). *For an expression e, result r, and closed strategy $s_\clubsuit$ we have:*

$$\frac{r \in [\![s_\clubsuit]\!]\xi e \qquad r \neq div}{e \xrightarrow{s_\clubsuit} r}$$

To prove this theorem, we first generalise to open strategies. To do this, we define an approximation relation between a closed strategy and an element of our domain, and state an approximation lemma. Here, we employ, for a simultaneous substitution $\theta : \mathbb{V} \to \mathbb{S}_\clubsuit$, the notation $s[\theta]$ for the application of θ to all free variables in s.

Definition 3.1 (Approximation relation one). *Given a closed strategy $s_\clubsuit$ and a function $d \in \mathfrak{D}$, we say $s_\clubsuit \triangle d$ if and only if for any given input expression e, when r is a non-diverging result obtained by applying d to e, r will also be the result of evaluating the big-step operational semantics of $s_\clubsuit$ with the input expression e.*

$$s_\clubsuit \triangle d \iff \forall e\ r.\ r \in d(e) \cap (\mathbb{E} \cup \{err\}) \Rightarrow e \xrightarrow{s_\clubsuit} r$$

Lemma 3.5 (Approximation lemma one).

$$\frac{\forall y \in fv(s).\ \theta(y) \triangle \xi(y) \qquad s_\clubsuit = s[\theta]}{s_\clubsuit \triangle [\![s]\!]\xi}$$

The proof of this lemma is by induction on the strategy s, and Scott induction is required for the fixed point cases. From the approximation lemma, we prove the computational adequacy theorem 3.4 by setting $s := s_\clubsuit$. As there are no free variables in $s_\clubsuit$, the approximation relation trivially implies our goal.

The computational soundness and adequacy theorems presented above state that the denotational semantics and big-step operational semantics are equivalent for the non-diverging cases. Next, we present computational soundness and adequacy theorems for divergent strategies.

The computational soundness theorem for the diverging cases states that, if the evaluation of the big-step operational semantics of a closed strategy $s_\clubsuit$ with an input expression e diverges, *div* must be in the resulting set obtained by executing the denotational semantics of $s_\clubsuit$ with the given expression e.

Theorem 3.6 (Computational soundness theorem two).

$$\frac{e \xrightarrow[\infty]{s_\clubsuit}}{div \in [\![s_\clubsuit]\!]\xi e}$$

Just as with computational adequacy for non-diverging cases, we must first generalise to open strategies. We define the second approximation relation together with an approximation lemma to prove this soundness theorem.

Definition 3.2 (Approximation relation two). *Given a closed strategy $s_\clubsuit$ and a function $d \in \mathfrak{D}$, we say $s_\clubsuit \ \triangle_\infty \ d$ if and only if for any given input expression e, when evaluating the big-step operational semantics of $s_\clubsuit$ with the input expression e diverges, the divergence div will be obtained by applying d to e, and we have the ordering $d(e) \preceq [\![s_\clubsuit]\!]\xi e$.*

$$s_\clubsuit \ \triangle_\infty \ d \iff \forall e.\ (e \xrightarrow[\infty]{s_\clubsuit} \Rightarrow div \in d(e)) \wedge d(e) \preceq [\![s_\clubsuit]\!]\xi e$$

Lemma 3.7 (Approximation lemma two).

$$\frac{\forall y \in fv(s).\ \theta(y) \ \triangle_\infty \ \xi(y) \qquad s_\clubsuit = s[\theta]}{s_\clubsuit \ \triangle_\infty \ [\![s]\!]\xi}$$

The proof of this lemma is (again) by induction on the strategy s, where Scott induction is used for the fixed point cases. For the cases which involve terminating sub-steps, such as sequential composition or left choice, we make use of our soundness theorem for non-diverging cases, theorem 3.2. We utilise this approximation lemma 3.7 to prove the computational soundness theorem 3.6.

Lastly, we prove the computational adequacy theorem for the diverging cases, which is again the converse of the soundness theorem 3.6. It states that if *div* is in a result of executing the denotational semantics of a closed strategy $s_\clubsuit$ with an input expression e, then evaluating the big-step operational semantics with the given expression e leads to divergence.

Theorem 3.8 (Computational adequacy theorem two).

$$\frac{div \in [\![s_\clubsuit]\!]\xi e}{e \xrightarrow[\infty]{s_\clubsuit}}$$

We prove this by *coinduction* over big-step operational semantics for diverging cases while making use of the computational adequacy theorem 3.4 for the non-diverging cases. Just as with our computational soundness proof for non-diverging cases, we work only with closed strategies $s_\clubsuit$, and rely on our substitution lemma 3.3 for the fixed point cases.

With these two pairs of computational soundness and adequacy theorems, we can conclude that the denotational semantics and big-step operational semantics are equivalent. Formally, we obtain:

Theorem 3.9 (Equivalence between semantics).

$$[\![s_\clubsuit]\!]\xi e = \{r \mid e \xrightarrow{s_\clubsuit} r\} \cup \{div \mid e \xrightarrow[\infty]{s_\clubsuit}\}$$

In this section, we have studied two styles of semantics of System S, namely a denotational semantics and a big-step operational semantics. To complete our semantic accounting, it may be worthwhile for us to study its small-step operational semantics in the future.

4 LOCATION-BASED WEAKEST PRECONDITION CALCULUS

As we have seen, a strategy either successfully rewrites an expression into another expression, generates an error, or fails to terminate.

Naturally, we care mainly about the *successful* executions of a strategy. In particular, when it rewrites an input expression into another expression that satisfies a desired property. In order to formally understand successful and unsuccessful executions of strategies, we design and formalise a location-based weakest precondition calculus. Weakest preconditions were introduced by Dijkstra [1975], as an axiomatic semantics for his guarded command language. Different from other weakest precondition calculi, we introduce the notion of a *location* in an AST as a parameter in our calculus for reasoning about traversals, which is discussed in section 4.1. Before presenting the formal definition of the calculus, we recapitulate the definition of a weakest precondition.

Definition 4.1 (Weakest precondition). *Given a program S and a postcondition P, the weakest precondition $wp(S, P)$ is an assertion R_w such that for any precondition R:*

$$\{R\}S\{P\} \Leftrightarrow (R \Rightarrow R_w)$$

Here $\{R\}S\{P\}$ is a Hoare triple stating that S will successfully terminate in a state satisfying assertion P if the state before executing S satisfies R. Intuitively, the weakest precondition of S under P characterises all those states that lead to successful termination in a state of P when executing S. In Dijkstra's [1975] calculus, a function *wp* is defined which, given a program and an assertion as a postcondition, computes the weakest precondition inductively on the program structure. Bonsangue and Kok [1992] extend Dijkstra's calculus to assign weakest preconditions for a fixed-point operator by additionally including a *logic environment* as an input to the *wp* function, which associates a predicate transformer with each variable. As we also have a fixed-point operator for general recursion, we do the same in this formalisation.

When dealing with strategies, assertions take the form of sets of expressions, and a state is an expression we are rewriting. Thus, the weakest precondition is a set of input expressions for a strategy to be applied to, such that the result of the application of the strategy will lead to another expression. That means the strategy will neither yield an error nor diverge. Moreover, the weakest precondition has to guarantee that an expression of the postcondition is reached.

Definition 4.2 (Weakest must succeed precondition). *A weakest must succeed precondition takes the form $wp_{\zeta \Vdash s@l}(P)$. This is the set of those expressions that, by applying strategy s at location l under the logic environment ζ, will be successfully transformed into expressions satisfying P.*

To calculate this set of input expressions constituting the weakest must succeed precondition, we also introduce the following auxiliary function. In fact, $wp_{\zeta \Vdash s@l}(P)$ and $wp^{\uparrow}_{\zeta \Vdash s@l}(P)$ will be defined by mutual induction.

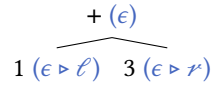
Definition 4.3 (Weakest may error precondition). *A weakest may error precondition takes the form of $wp^{\uparrow}_{\zeta \Vdash s@l}(P)$. This is the set of those expressions that, by applying strategy s at location l under the logic environment ζ, will be successfully transformed into expressions satisfying P,* ***or result in error****.*

4.1 Modelling Traversals

In definitions 4.2 and 4.3, we introduce the location for specifying the particular sub-expression to which the strategy s should be applied. This allows us to express that after applying a strategy s to the sub-expression *at the location* l of an input expression e, the input expression e should be transformed into an expression that satisfies the postcondition P. Consequently, the weakest precondition for traversals such as $one(s)$, $some(s)$, and $all(s)$ can be defined inductively in terms of the weakest precondition of s, just at different locations.

Kieburtz [2001] proposes an alternative approach, using modal logic for assertions about traversals. However, it is unclear how this technique could be used to define a complete calculus. We discuss this in section 6.

A *location* is essentially a path into the abstract syntax tree. Such a path consists of a sequence of positions, for our binary trees either left (ℓ) or right (r). Positions are prepended to a location with $\triangleleft$ and appended with $\triangleright$. For instance, suppose we have an AST representing an arithmetic expression 1 + 3, each sub-expression is located as:



With locations being introduced in the assertions, accompanied by the two helper functions *lookup* and *update* discussed in the next section, we can model the execution of a strategy at a given location in the input expression, which enables the assignments of weakest preconditions inductively for traversals just as with other operators.

4.2 The Calculus

We now introduce the location-based weakest precondition calculus for System S in its full formal detail. We first provide definitions of helper functions and essential notations for the formalisation.

To connect locations and expressions, we introduce two partial functions *lookup* and *update*, shown in figure 7. Given a location l and an expression e, the partial function *lookup* returns the sub-expression which is located at the location l in an expression e. The function is partial, as it is only defined when the location l actually exists in the expression e. The partial function *update* takes in a set $xs \in \mathfrak{D}_p$, and updates an expression e at the location l with each expression in xs, resulting in a set of expressions where each element is obtained by replacing the sub-expression of e at the location l with an element of xs, with appropriate handling of errors and divergence.

Figure 8 shows the essential notations for defining the weakest precondition calculus. Since we will again have fixed-point operators in the weakest precondition calculus, we need to ensure that least fixed points exist, by operating in a domain which is again a cpo, and show that our wp function is monotone with respect to that domain. The ordering of our domain $\mathfrak{D}_L$ is a point-wise lifted set ordering, of which the bottom element is the empty set.

Similar to the semantic environment introduced for the denotational semantics in figure 4, the logic environment contains mappings of (fixed point) variables to an element of our logic domain (which is a function). Since we mutually define weakest must succeed preconditions and weakest may error preconditions, a fixed-point variable can map to two different functions. We use the tags $\cdot$ (must succeed) and $\uparrow$ (may error) to distinguish these two different mappings.

With these notations and helper partial functions, we provide the location-based weakest precondition calculus. For presentation purposes, we simplify our definitions by only considering the

$$lookup : \mathbb{L} \to \mathbb{E} \rightharpoonup \mathbb{E} \quad (\text{We write it as } \pitchfork_{l:\mathbb{L}}\ (e : \mathbb{E}) : (e' : \mathbb{E}))$$

$$lookup\ \epsilon\ e = e$$

$$lookup\ (\ell \triangleleft l)\ \overset{n}{\underset{e_1 \quad e_2}{\wedge}} = lookup\ l\ e_1$$

$$lookup\ (r \triangleleft l)\ \overset{n}{\underset{e_1 \quad e_2}{\wedge}} = lookup\ l\ e_2$$

$$update : \mathbb{L} \to \mathbb{E} \rightharpoonup \mathfrak{D}_p \to \mathfrak{D}_p \quad (\text{We write it as } (d : \mathfrak{D}_p) \boxdot\!\!\Rightarrow_{l:\mathbb{L}}\ (e : \mathbb{E}) : (d' : \mathfrak{D}_p))$$

$$update\ \epsilon\ e\ xs = xs$$

$$update\ (\ell \triangleleft l)\ \overset{n}{\underset{e_1 \quad e_2}{\wedge}}\ xs = \{\ \overset{n}{\underset{e_1' \quad e_2}{\wedge}}\ \mid e_1' \in (update\ l\ e_1\ xs) \cap \mathbb{E}\} \cup (xs \cap \{err, div\})$$

$$update\ (r \triangleleft l)\ \overset{n}{\underset{e_1 \quad e_2}{\wedge}}\ xs = \{\ \overset{n}{\underset{e_1 \quad e_2'}{\wedge}}\ \mid e_2' \in (update\ l\ e_2\ xs) \cap \mathbb{E}\} \cup (xs \cap \{err, div\})$$

Fig. 7. Helper functions

$$\begin{array}{rlrl} \text{Position} & i ::= \ell \mid r & \text{Location}(\mathbb{L}) & l ::= \epsilon \mid l \triangleright i \mid i \triangleleft l \\ \text{Variable}(\mathbb{V}) & X\ Y\ Z \ldots & \text{Tag}(\mathbb{T}) & t ::= \cdot \mid \uparrow \end{array}$$

$$\begin{array}{rl} \text{Logic Domain} & \mathfrak{D}_L = \mathbb{L} \to \mathscr{P}(\mathbb{E}) \to \mathscr{P}(\mathbb{E}) \\ \text{Logic Environment}(\Gamma_L) & \zeta : (\mathbb{V} \times \mathbb{T}) \to \mathfrak{D}_L \end{array}$$

$$wp_{\zeta:\Gamma_L \Vdash s:\mathbb{S}@l:\mathbb{L}}(P : \mathscr{P}(\mathbb{E})) : (R_w : \mathscr{P}(\mathbb{E})) \qquad \text{(Weakest must succeed precondition)}$$

$$wp^{\uparrow}_{\zeta:\Gamma_L \Vdash s:\mathbb{S}@l:\mathbb{L}}(P : \mathscr{P}(\mathbb{E})) : (R_w : \mathscr{P}(\mathbb{E})) \qquad \text{(Weakest may error precondition)}$$

Fig. 8. Basic notations

cases where location l actually exists in the expression. In our Isabelle/HOL formalisation, we make this explicit in the definition of wp and $wp^{\uparrow}$.

Figure 9 shows the weakest preconditions for basic strategies: SKIP, ABORT and *atomic*. Trivially, the weakest must succeed precondition and weakest may error precondition for SKIP are the same, i.e., the given postcondition P, since the execution of SKIP never results in error or divergence, nor changes the input expression. As for ABORT, since it will always result in an error no matter what input expression is given, its weakest must succeed precondition is the empty set and its weakest may error precondition is the set of all expressions. The weakest preconditions of atomic strategies are defined using their denotational semantics (cf. figure 4): the weakest must succeed precondition is the set of input expressions, for each expression of which applying the atomic strategy to its sub-expression at the given location l should result in a (singleton) set of expressions which is a subset of the given postcondition P. The weakest may error postcondition is defined in a similar manner, the only difference is that the resulting set of expressions should be a subset of $P \cup \{err\}$. It does not matter what semantic environment is given here when we invoke the semantics, so we just use the environment which maps all variables to $\{div\}$, denoted by $\varnothing$. Remember that the operators $\pitchfork$ and $\boxdot\Longrightarrow$ are *lookup* and *update*.

Figure 10 shows the weakest preconditions for combinators: sequential composition, left choice and nondeterministic choice. Intuitively, the weakest must succeed precondition of the sequential composition $s\ ;\ t$ is simply to sequentially compose the weakest must succeed preconditions of s and t where the post condition of s is the weakest must succeed precondition of t. The same approach is taken for defining the weakest may error precondition. The weakest must succeed precondition of the left choice $s <+ t$ is the union of the set of expressions that can be successfully rewritten by the strategy s and the set of expressions for which applying s may result in error but that can be successfully rewritten by the strategy t. Its weakest may error condition additionally includes the set of expressions for which applying the strategy t may result in error. The definitions of the

$$wp_{\zeta \Vdash \text{SKIP}@l}(P) = P \qquad wp_{\zeta \Vdash \text{ABORT}@l}(P) = \emptyset$$

$$wp^{\uparrow}_{\zeta \Vdash \text{SKIP}@l}(P) = P \qquad wp^{\uparrow}_{\zeta \Vdash \text{ABORT}@l}(P) = \mathbb{E}$$

$$wp_{\zeta \Vdash atomic@l}(P) = \{e \mid (\llbracket atomic \rrbracket \varnothing(\pitchfork_l\ e))\ \boxdot\Longrightarrow_l\ e \subseteq P\}$$

$$wp^{\uparrow}_{\zeta \Vdash atomic@l}(P) = \{e \mid (\llbracket atomic \rrbracket \varnothing(\pitchfork_l\ e))\ \boxdot\Longrightarrow_l\ e \subseteq P \cup \{err\}\}$$

Fig. 9. Location-based weakest preconditions for basic strategies

$$wp_{\zeta \Vdash s\,;\,t@l}(P) = wp_{\zeta \Vdash s@l}(wp_{\zeta \Vdash t@l}(P)) \qquad wp^{\uparrow}_{\zeta \Vdash s\,;\,t@l}(P) = wp^{\uparrow}_{\zeta \Vdash s@l}(wp^{\uparrow}_{\zeta \Vdash t@l}(P))$$

(Sequential composition)

$$wp_{\zeta \Vdash s<+t@l}(P) = wp_{\zeta \Vdash s@l}(P) \cup (wp^{\uparrow}_{\zeta \Vdash s@l}(P) \cap wp_{\zeta \Vdash t@l}(P))$$
$$wp^{\uparrow}_{\zeta \Vdash s<+t@l}(P) = wp_{\zeta \Vdash s@l}(P) \cup (wp^{\uparrow}_{\zeta \Vdash s@l}(P) \cap wp^{\uparrow}_{\zeta \Vdash t@l}(P))$$

(Left choice)

$$wp_{\zeta \Vdash s<+>t@l}(P) = (wp^{\uparrow}_{\zeta \Vdash t@l}(P) \cap wp_{\zeta \Vdash s@l}(P)) \cup (wp^{\uparrow}_{\zeta \Vdash s@l}(P) \cap wp_{\zeta \Vdash t@l}(P))$$
$$wp^{\uparrow}_{\zeta \Vdash s<+>t@l}(P) = wp^{\uparrow}_{\zeta \Vdash s@l}(P) \cap wp^{\uparrow}_{\zeta \Vdash t@l}(P)$$

(Nondeterministic choice)

Fig. 10. Location-based weakest preconditions for combinators

weakest preconditions of the nondeterministic choice s <+> t capture the angelic nondeterminism for *err* and demonic nondeterminism for *div*. Its weakest must succeed precondition is the set of expressions to which applying neither the strategy s nor t will diverge and which can be successfully rewritten by at least one of s and t. The weakest may error precondition relaxes this last requirement by including the set of expressions to which applying both s and t may result in an error.

Location is very important for defining the weakest preconditions of traversals. Demonstrated in figure 11, the approach of defining the weakest preconditions for $one(s)$ is again similar to nondeterministic choice, as $one(s)$ nondeterministically chooses one of the left or right child of the current expression to apply the strategy s to. Its weakest must succeed precondition is a set of expressions that are not leaf nodes. For each of them, applying s to either its left child or right child should not diverge, and at least one of its children must be successfully rewritten by s. The weakest may error precondition of $one(s)$ includes all expressions that are leaf nodes as well as expressions whose both children to which applying s may result in error. The weakest must succeed precondition of $some(s)$ is a set of expressions that are not leaf nodes. For each of them, if the given strategy s can be applied to *both* of its children successfully, the result of applying s to both of them regardless of the ordering of the application should satisfy the postcondition P. In addition, applying s to one of its children may result in an error, but not for both of its children. Again, expressions with children to which applying s diverges are excluded from the weakest must succeed precondition. Similar to $one(s)$, the weakest may error preconditions includes all leaf expressions and expressions whose both children to which applying s may result in error. Since $all(s)$ requires the strategy s to be applied to either a leaf expression or both children of an expression which is not a leaf, intuitively, its weakest must succeed precondition is a set of leaf expressions, or expressions of which both children can be successfully rewritten by the strategy s regardless of the order of the application of s. Its weakest may error precondition again includes all leaf expressions and expressions with children to which applying s may result in an error.

Lastly, we introduce the weakest preconditions for the fixed-point operator, shown in figure 12, which are defined using simultaneous induction. Δ contains a pair of simultaneously defined least fixed points $\mathscr{X}$ and $\mathscr{Y}$ which are used to define the weakest must succeed precondition and weakest may fail precondition respectively. In these fixed-point equations, we extend the logic environment ζ with mappings from the fixed-point variable with tags $(X, \cdot)$ and $(X, \uparrow)$ to the least fixed points $\mathscr{X}$ and $\mathscr{Y}$ respectively.

$$wp_{\zeta \Vdash one(s)@l}(P) = (wp^{\uparrow}_{\zeta \Vdash s@l \triangleright \ell}(P) \cap wp_{\zeta \Vdash s@l \triangleright r}(P)) \cup (wp^{\uparrow}_{\zeta \Vdash s@l \triangleright r}(P) \cap wp_{\zeta \Vdash s@l \triangleright \ell}(P))$$
$$wp^{\uparrow}_{\zeta \Vdash one(s)@l}(P) = \{e \mid (\pitchfork_l\ e) = Leaf\} \cup (wp^{\uparrow}_{\zeta \Vdash s@l \triangleright \ell}(P) \cap wp^{\uparrow}_{\zeta \Vdash s@l \triangleright r}(P))$$

(One)

$$\begin{aligned} wp_{\zeta \Vdash some(s)@l}(P) = {} & wp_{\zeta \Vdash s@l \triangleright \ell}(wp_{\zeta \Vdash s@l \triangleright r}(P)) \cup wp_{\zeta \Vdash s@l \triangleright r}(wp_{\zeta \Vdash s@l \triangleright \ell}(P)) \\ & \cup (wp_{\zeta \Vdash s@l \triangleright \ell}(P) \cap wp_{\zeta \Vdash s@l \triangleright \ell}(wp^{\uparrow}_{\zeta \Vdash s@l \triangleright r}(P))) \\ & \cup (wp_{\zeta \Vdash s@l \triangleright r}(P) \cap wp_{\zeta \Vdash s@l \triangleright r}(wp^{\uparrow}_{\zeta \Vdash s@l \triangleright \ell}(P))) \end{aligned}$$

$$\begin{aligned} wp^{\uparrow}_{\zeta \Vdash some(s)@l}(P) = {} & \{e \mid (\pitchfork_l\ e) = Leaf\} \\ & \cup wp^{\uparrow}_{\zeta \Vdash s@l \triangleright \ell}(wp^{\uparrow}_{\zeta \Vdash s@l \triangleright r}(P)) \cap wp^{\uparrow}_{\zeta \Vdash s@l \triangleright r}(wp^{\uparrow}_{\zeta \Vdash s@l \triangleright \ell}(P)) \\ & \cap (wp^{\uparrow}_{\zeta \Vdash s@l \triangleright \ell}(P) \cup wp^{\uparrow}_{\zeta \Vdash s@l \triangleright \ell}(wp_{\zeta \Vdash s@l \triangleright r}(P))) \\ & \cap (wp^{\uparrow}_{\zeta \Vdash s@l \triangleright r}(P) \cup wp^{\uparrow}_{\zeta \Vdash s@l \triangleright r}(wp_{\zeta \Vdash s@l \triangleright \ell}(P))) \end{aligned}$$

(Some)

$$\begin{aligned} wp_{\zeta \Vdash all(s)@l}(P) = {} & (P \cap \{e \mid (\pitchfork_l\ e) = Leaf\}) \\ & \cup wp_{\zeta \Vdash s@l \triangleright \ell}(wp_{\zeta \Vdash s@l \triangleright r}(P)) \cup wp_{\zeta \Vdash s@l \triangleright r}(wp_{\zeta \Vdash s@l \triangleright \ell}(P)) \end{aligned}$$
$$\begin{aligned} wp^{\uparrow}_{\zeta \Vdash all(s)@l}(P) = {} & (P \cap \{e \mid (\pitchfork_l\ e) = Leaf\}) \\ & \cup (wp^{\uparrow}_{\zeta \Vdash s@l \triangleright \ell}(wp^{\uparrow}_{\zeta \Vdash s@l \triangleright r}(P)) \cap wp^{\uparrow}_{\zeta \Vdash s@l \triangleright r}(wp^{\uparrow}_{\zeta \Vdash s@l \triangleright \ell}(P))) \end{aligned}$$

(All)

Fig. 11. Location-based weakest preconditions for traversals

$$wp_{\zeta \Vdash X@l}(P) = \zeta(X, \cdot)\, l\, P \ \text{ (where } \zeta(X, \cdot) \textbf{ def.)} \quad wp^{\uparrow}_{\zeta \Vdash X@l}(P) = \zeta(X, \uparrow)\, l\, P \ \text{ (where } \zeta(X, \uparrow) \textbf{ def.)}$$

(Fixed-point variable)

$$wp_{\zeta \Vdash \mu X.s@l}(P) = [\mathrm{LFP}\ \mathscr{X} : \Delta]\, l\, P \qquad wp^{\uparrow}_{\zeta \Vdash \mu X.s@l}(P) = [\mathrm{LFP}\ \mathscr{Y} : \Delta]\, l\, P$$
$$\textbf{Where:} \quad \Delta = \begin{cases} \mathscr{X}\, l\, P & = wp_{\zeta[(X,\cdot) \mapsto \mathscr{X},\ (X,\uparrow) \mapsto \mathscr{Y}] \Vdash s@l}(P) \\ \mathscr{Y}\, l\, P & = wp^{\uparrow}_{\zeta[(X,\cdot) \mapsto \mathscr{X},\ (X,\uparrow) \mapsto \mathscr{Y}] \Vdash s@l}(P) \end{cases}$$

(Fixed-point operator)

Fig. 12. Location-based weakest preconditions for fixed-point operators

The weakest must succeed precondition a (fixed point) variable X is calculated by applying the function obtained by looking up $(X, \cdot)$ in the logic environment ζ to the location l and postcondition P. For the weakest may fail precondition, the function applied to l and P is obtained by looking up X with the may fail tag $\uparrow$ from ζ.

4.3 The Soundness of the Weakest Precondition Calculus w.r.t. the Formal Semantics

Since our weakest precondition calculus is designed to reason about the execution of strategies, it is essential to prove it is *sound* with respect to the formal semantics introduced in section 3. Specifically, we define the soundness of the weakest must succeed precondition as theorem 4.1, and the soundness of the weakest may error precondition as theorem 4.2. Both of these theorems have the same assumption to relate the logic and semantic environments ζ and ξ. This assumption states that given any variable X, location l and postcondition P, executing the function obtained by looking

up X in the logic environment ζ — with the must succeed tag or the may error tag correspondingly — gives the set of expressions, at the location l of each of which executing the semantics of the variable ($\xi(X)$) results in a subset of the postcondition P or $P \cup \{err\}$ respectively. From this assumption, theorem 4.1 concludes that the weakest must succeed precondition $wp_{\zeta \Vdash s@l}(P)$ should equal to the set of expressions on which executing the semantics of s gives a subset of P. Similarly, theorem 4.2 says that under the same assumptions, the weakest may error precondition $wp^{\uparrow}_{\zeta \Vdash s@l}(P)$ should equal to the set of expressions on which executing the semantics of s gives a subset of $P \cup \{err\}$.

Theorem 4.1 (Soundness theorem for Weakest Must Succeed Precondition).

$$\frac{\begin{array}{c}\forall X\, l\, P.\, \zeta(X,\cdot)\ l\ P = \{e \mid \xi(X)(\pitchfork_l\ e)\ \square\!\Rightarrow_l\ e \subseteq P\} \\ \wedge \zeta(X,\uparrow)\ l\ P = \{e \mid \xi(X)(\pitchfork_l\ e)\ \square\!\Rightarrow_l\ e \subseteq P \cup \{err\}\}\end{array}}{wp_{\zeta \Vdash s@l}(P) = \{e \mid (\llbracket s \rrbracket \xi(\pitchfork_l\ e))\ \square\!\Rightarrow_l\ e \subseteq P\}}$$

Theorem 4.2 (Soundness theorem for Weakest May Error Precondition).

$$\frac{\begin{array}{c}\forall X\, l\, P.\, \zeta(X,\cdot)\ l\ P = \{e \mid \xi(X)(\pitchfork_l\ e)\ \square\!\Rightarrow_l\ e \subseteq P\} \\ \wedge \zeta(X,\uparrow)\ l\ P = \{e \mid \xi(X)(\pitchfork_l\ e)\ \square\!\Rightarrow_l\ e \subseteq P \cup \{err\}\}\end{array}}{wp^{\uparrow}_{\zeta \Vdash s@l}(P) = \{e \mid (\llbracket s \rrbracket \xi(\pitchfork_l\ e))\ \square\!\Rightarrow_l\ e \subseteq P \cup \{err\}\}}$$

We prove these two theorems simultaneously, by induction on the strategy s. For the fixed-point operator cases, we make use of Scott induction. The proof is mechanised in Isabelle/HOL.

5 REASONING ABOUT STRATEGIES WITH WEAKEST PRECONDITION CALCULUS

As discussed in section 2, there are some strategies that can never be executed successfully, such as strategies that always diverge like *repeat*(SKIP) and strategies that are not well composed like $mult_{com}$; add_{com}. We call such strategies *bad* strategies. Formally, we define *good* and *bad* strategies in terms of our weakest precondition calculus as definition 5.1 and definition 5.2, where the formal definition of bad strategies is the negation of good strategies.

Definition 5.1 (Good strategies). *A strategy s is good iff for a given postcondition P:*

$$wp_{\zeta \Vdash s@l}(P) \neq \emptyset$$

Definition 5.2 (Bad strategies). *A strategy s is bad iff for a given postcondition P:*

$$wp_{\zeta \Vdash s@l}(P) = \emptyset$$

For strategies that can terminate and are well composed, they may not be able to successfully rewrite any input expression into an expression satisfying our desired postcondition. For instance, even though the atomic strategy add_{com} is a good strategy, applying it to $3 * 4$ would result in an error. Also, as illustrated in section 2, when encoding a normalisation strategy for rewriting an input lambda expression into its $\beta\eta$-normal form, such strategy can diverge on some input expressions (e.g., the expression Ω given below). If it does terminate on an input expression, it ought to rewrite all reducible sub-expressions of such input expression. We formally define the successful executions and unsuccessful executions of good strategies as definition 5.3 and definition 5.4.

Definition 5.3 (Successful execution). *An execution of a good strategy s, on an input expression e is successful iff for a given postcondition P:*

$$e \in wp_{\zeta \Vdash s@l}(P) \qquad \text{(where:} \quad wp_{\zeta \Vdash s@l}(P) \neq \emptyset\text{)}$$

Definition 5.4 (Unsuccessful execution). *An execution of a good strategy s on an input expression e is unsuccessful iff for a given postcondition P:*

$$e \notin wp_{\zeta \Vdash s @ l}(P) \qquad \text{(where:} \quad wp_{\zeta \Vdash s @ l}(P) \neq \emptyset)$$

Next, we demonstrate how to use the location-based weakest precondition calculus to reason about the execution of strategies. All examples we discuss are mechanised in Isabelle/HOL.

5.1 Reasoning About Termination

Strategies can diverge. Recall from section 2 that $repeat(s)$ is defined as $\mu X.try(s; X)$ where $try(s)$ is defined as s <+ SKIP. We can derive the weakest precondition formula of $repeat(s)$ by the weakest precondition formulae of SKIP, left choice, sequential composition and the fixed-point operator:

$$wp_{\zeta \Vdash repeat(s) @ l}(P) = wp^{\uparrow}_{\zeta \Vdash repeat(s) @ l}(P) = [\text{LFP } \mathcal{X} : \Delta]\ l\ P$$

where Δ is the fixed-point equation

$$\mathcal{X}\ l\ P = wp_{\zeta[(X,\cdot)\mapsto\mathcal{X}\ ,\ (X,\uparrow)\mapsto\mathcal{X}] \Vdash s @ l}(\mathcal{X}\ l\ P) \cup (P \cap wp^{\uparrow}_{\zeta[(X,\cdot)\mapsto\mathcal{X}\ ,\ (X,\uparrow)\mapsto\mathcal{X}] \Vdash s @ l}(\mathcal{X}\ l\ P))$$

Although the execution of $repeat(s)$ would never result in an error since its weakest may error precondition formula is identical to its weakest must succeed precondition, it may diverge.

A simple example of a diverging strategy we have introduced is the strategy $repeat$(SKIP). It is straightforward to conclude that it is a bad strategy using the weakest precondition calculus. With the weakest must succeed precondition formulae of $repeat(s)$ and SKIP, we calculate that for the set of all expressions as the post condition, the weakest must succeed precondition of $repeat$(SKIP) is an empty set:

$$wp_{\zeta \Vdash repeat(\text{SKIP}) @ \epsilon}(\mathbb{E}) = \emptyset$$

Intuitively, such a result indicates that there is no expression that can be successfully rewritten by the strategy $repeat$(SKIP). According to the definition 5.2, we can conclude that the diverging strategy $repeat$(SKIP) is bad strategy.

Since we apply demonic nondeterminism on divergence as discussed in section 4, the strategy SKIP <+> $repeat$(SKIP) always diverges. To show that it is a bad strategy, we can again calculate its weakest must succeed precondition with the set of all expressions as the postcondition:

$$wp_{\zeta \Vdash \text{SKIP} <+> repeat(\text{SKIP}) @ \epsilon}(\mathbb{E}) = \emptyset$$

Again, we obtain an empty set as its weakest must succeed precondition, indicating that such a strategy can never be successfully executed on any input expression.

Strategies that can terminate are potentially good strategies. For instance, the strategy SKIP <+ $repeat$(SKIP) always terminates. To conclude it being a good strategy, we calculate its weakest must succeed precondition:

$$wp_{\zeta \Vdash \text{SKIP} <+ repeat(\text{SKIP}) @ \epsilon}(\mathbb{E}) = \mathbb{E}$$

Intuitively, because left choice prioritises the strategy on the left hand side of the combinator over the strategy on the right hand side, SKIP is always preferred over $repeat$(SKIP) here. Therefore, SKIP <+ $repeat$(SKIP) always terminates and produces expressions. According to the definition 5.1, we conclude that the terminating strategy SKIP <+ $repeat$(SKIP) is a good strategy.

5.2 Reasoning About Well Composed Strategies

Strategies that terminate may still not be good strategies, since they can be not well composed and always result in error. An example of a not well composed strategy that we have introduced in section 2 is $mult_{com}$; add_{com}. According to the weakest precondition formulae for atomic strategies

$$\text{Lambda Expression} \quad e := \quad Id\ \iota \ \mid \ \overset{Abs}{\underset{\bullet \quad e}{\wedge}} \ \mid \ \overset{App}{\underset{e \quad e}{\wedge}}$$

$$\text{Index} \quad \iota \ \in \ \mathbb{N}$$

Fig. 13. The syntax of the lambda calculus

and the sequential composition presented in figure 9 and figure 10, we calculate its weakest must succeed precondition for the set of all expressions as the postcondition:

$$wp_{\zeta \Vdash mult_{com}\,;\, add_{com}@\epsilon}(\mathbb{E}) = \emptyset$$

Since its weakest must succeed precondition is an empty set, with definition 5.2, we can conclude that the strategy $mult_{com}$; add_{com} is a bad strategy.

Well composed terminating strategies are good strategies. For example, given an atomic strategy add_{id} defined as:

$$add_{id} : 0 + a \rightsquigarrow a$$

The strategy add_{com} ; add_{id} is a well composed strategy. In practice, it can successfully rewrite an expression $3 + 0$ into the expression 3. We are able to conclude that the strategy add_{com} ; add_{id} is a good strategy again by checking its weakest must succeed precondition for the set of all expressions as the postcondition:

$$wp_{\zeta \Vdash add_{com}\,;\, add_{id}@\epsilon}(\mathbb{E}) = \{e \mid e = a + 0\}$$

Since the calculated weakest must succeed precondition is not an empty set, according to the definition 5.1, the strategy add_{com} ; add_{id} is a good strategy.

5.3 Reasoning About Beta-Eta Normalisation

In section 2, we have defined the normalise strategy by composing the strategy $repeat(s)$ and the top-down traversal $topDown(s)$ as $normalise(s) = repeat(topDown(s))$, which keeps applying a given strategy s to every possible sub-expression of an expression until s is no longer applicable.

One example usage of the normalisation strategy we demonstrated is to reduce an expression in λ-calculus into the $\beta\eta$-normal form. Given the β-reduction and η-reduction as two atomic strategies *beta* and *eta*, we can express the strategy for calculating the $\beta\eta$-normal form as:

$$BENF = normalise(beta <+\ eta)$$

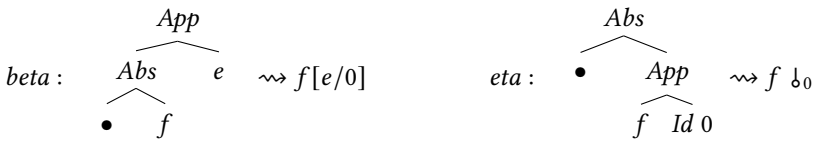
Furthermore, we define a predicate to assert that an expression is in $\beta\eta$-normal form, simply by stating that the *beta* and *eta* atomic strategies must not be defined for any location in the expression:

$$isBENF\ e \Leftrightarrow \forall l.\ beta(\pitchfork_l\ e)\ \textbf{undef} \wedge eta(\pitchfork_l\ e)\ \textbf{undef} \qquad (\text{where:} \quad \pitchfork_l\ e \text{ is defined})$$

It is well known that not every λ-expression has such a normal form. With our location-based weakest precondition calculus, we are able to reason about whether an expression can be normalised by the strategy *BENF* into a $\beta\eta$-normal form.

Firstly, in figure 13, we provide an encoding of the lambda calculus with de Bruijn indices using the expression tree structure we introduced, which takes the form of either a *Leaf* or a node $\overset{n}{\underset{e \quad e}{\wedge}}$. Specifically, we encode an *Id* expression (a de Bruijn index) as a *Leaf* and both an abstraction and

an application as nodes. Then we encode beta reduction and eta reduction as two atomic strategies:



where $f[e/0]$ is the de Bruijn substitution of the index 0 with the expression e in f and $f\downarrow_0$ is the de Bruijn down shifting eliminating the index 0 in f.

Next we introduce the weakest precondition formula for the strategy *normalise*(s), which is calculated using the weakest precondition formulae of *repeat*(s) (introduced in section 5.1) and *topDown*(s). Recall that in section 2 the strategy *topDown*(s) is defined using the left choice combinator, the traversal *one*(s) as well as the fixed-point operator:

$$topDown(s) = \mu X.(s <+\ one(X))$$

We can derive its weakest must succeed precondition and weakest may error precondition formulae:

$$wp_{\zeta \Vdash topDown(s)@l}(P) = [\mathrm{LFP}\ \mathcal{X} : \Delta]\ l\ P \qquad wp^{\uparrow}_{\zeta \Vdash topDown(s)@l}(P) = [\mathrm{LFP}\ \mathcal{Y} : \Delta]\ l\ P$$

Where:

$$\Delta = \begin{cases} \mathcal{X}\ l\ P & = wp_{\zeta[(X,\cdot)\mapsto\mathcal{X}\ ,\ (X,\uparrow)\mapsto\mathcal{Y}] \Vdash s@l}(P) \cup (wp^{\uparrow}_{\zeta[(X,\cdot)\mapsto\mathcal{X}\ ,\ (X,\uparrow)\mapsto\mathcal{Y}] \Vdash s@l}(P) \\ & \cap ((\mathcal{Y}\ (l \triangleright \ell)\ P \cap \mathcal{X}\ (l \triangleright \nearrow)\ P) \cup (\mathcal{Y}\ (l \triangleright \nearrow)\ P \cap \mathcal{X}\ (l \triangleright \ell)\ P))) \\ \mathcal{Y}\ l\ P & = wp_{\zeta[(X,\cdot)\mapsto\mathcal{X}\ ,\ (X,\uparrow)\mapsto\mathcal{Y}] \Vdash s@l}(P) \cup (wp^{\uparrow}_{\zeta[(X,\cdot)\mapsto\mathcal{X}\ ,\ (X,\uparrow)\mapsto\mathcal{Y}] \Vdash s@l}(P) \\ & \cap (\mathcal{Y}\ (l \triangleright \ell)\ P \cap \mathcal{Y}\ (l \triangleright \nearrow)\ P)) \end{cases}$$

With the weakest precondition formulae for *topDown*(s) defined, we can subsequently provide the weakest precondition formula for the strategy *normalise*(s). Note that its weakest must succeed precondition and weakest may error precondition share the same formula, just like *repeat*(s):

$$wp_{\zeta \Vdash normalise(s)@l}(P) = wp^{\uparrow}_{\zeta \Vdash normalise(s)@l}\zeta(P) = [\mathrm{LFP}\ \mathcal{X}_r : \Delta_r]\ l\ P$$

Where:

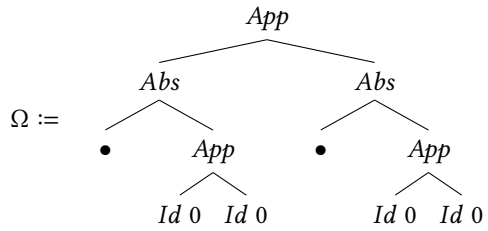
$$\Delta_r = \mathcal{X}_r\ l\ P = [\mathrm{LFP}\ \mathcal{X}_t : \Delta_t]\ l\ P \cup (([\mathrm{LFP}\ \mathcal{Y}_t : \Delta_t]\ l\ P) \cap P)$$

$$\Delta_t = \begin{cases} \mathcal{X}_t\ l\ P & = wp_{\zeta[(X,\cdot)\mapsto\mathcal{X}_r\ ,\ (X,\uparrow)\mapsto\mathcal{X}_r\ ,\ (Y,\cdot)\mapsto\mathcal{X}_t\ ,\ (Y,\uparrow)\mapsto\mathcal{Y}_t] \Vdash s@l}(\mathcal{X}_r\ l\ P) \\ & \cup (wp^{\uparrow}_{\zeta[(X,\cdot)\mapsto\mathcal{X}_r\ ,\ (X,\uparrow)\mapsto\mathcal{X}_r\ ,\ (Y,\cdot)\mapsto\mathcal{X}_t\ ,\ (Y,\uparrow)\mapsto\mathcal{Y}_t] \Vdash s@l}(\mathcal{X}_r\ l\ P) \\ & \cap ((\mathcal{Y}_t\ (l \triangleright \ell)\ P \cap \mathcal{X}_t\ (l \triangleright \nearrow)\ P) \cup (\mathcal{Y}_t\ (l \triangleright \nearrow)\ P \cap \mathcal{X}_t\ (l \triangleright \ell)\ P))) \\ \mathcal{Y}_t\ l\ P & = wp_{\zeta[(X,\cdot)\mapsto\mathcal{X}_r\ ,\ (X,\uparrow)\mapsto\mathcal{X}_r\ ,\ (Y,\cdot)\mapsto\mathcal{X}_t\ ,\ (Y,\uparrow)\mapsto\mathcal{Y}_t] \Vdash s@l}(\mathcal{X}_r\ l\ P) \\ & \cup (wp^{\uparrow}_{\zeta[(X,\cdot)\mapsto\mathcal{X}_r\ ,\ (X,\uparrow)\mapsto\mathcal{X}_r\ ,\ (Y,\cdot)\mapsto\mathcal{X}_t\ ,\ (Y,\uparrow)\mapsto\mathcal{Y}_t] \Vdash s@l}(\mathcal{X}_r\ l\ P) \\ & \cap (\mathcal{Y}_t\ (l \triangleright \ell)\ P \cap \mathcal{Y}_t\ (l \triangleright \nearrow)\ P)) \end{cases}$$

With the weakest precondition formula for *normalise*(s), we can first conclude that the strategy *BENF* for calculating the $\beta\eta$-normal form for expressions is a good strategy by showing:

$$wp_{\zeta \Vdash BENF@l}(\mathbb{E}) \neq \emptyset$$

Although the strategy *BENF* is good, some expressions are not able to be rewritten by it to a $\beta\eta$-normal form. For instance, the expression Ω is defined as:

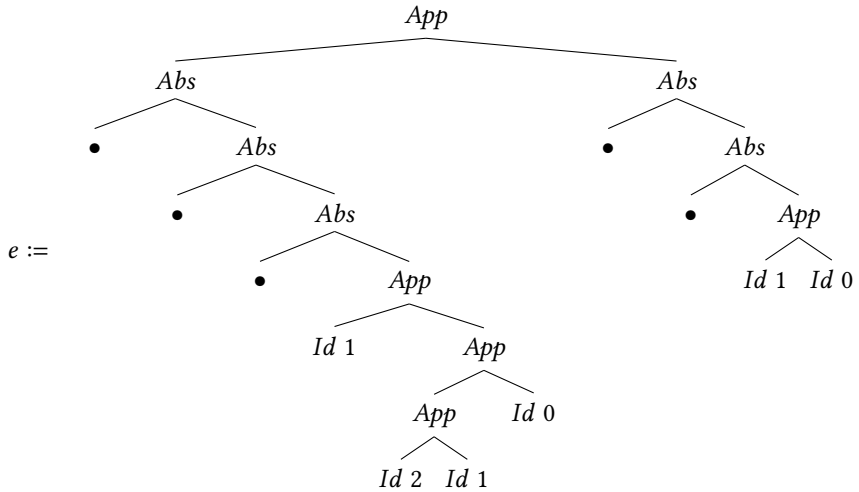


Applying the strategy *BENF* to the expression Ω will diverge, namely, the execution of the strategy *BENF* on Ω is unsuccessful. We draw this conclusion by showing that Ω is not an expression in the weakest must succeed precondition of *BENF* no matter what the postcondition is:

$$\Omega \notin wp_{\zeta \Vdash BENF@\epsilon}(\mathbb{E})$$

We prove this proposition straightforwardly using Scott induction.

Beside identifying expressions that fail to be normalised into a $\beta\eta$-normal form using *BENF*, we are also interested in examining whether a complex expression is indeed rewritten into a $\beta\eta$-normal form after applying the strategy *BENF*. For instance, given an expression e defined as:



we show that applying the strategy *BENF* to the expression e does rewrite it to a $\beta\eta$-normal form by showing the proposition below holds:

$$e \in wp_{\zeta \Vdash BENF@\epsilon}(\{e \mid isBENF\ e\})$$

The proof of this proposition is also straightforward, merely requiring the repeated unfolding of fixed-point operators. On the basis of this result, we can conclude that the strategy *BENF* performs the rewrite on the input expression e as we expected, namely, rewriting e into its $\beta\eta$-normal form.

5.4 Discussion

As this section demonstrates, our formal calculus provides precise description of strategies, independent of their length and complexity. It also provides a good characterisation of desired properties to be satisfied after the execution of a strategy, as well as of expressions that can be

successfully rewritten. Additionally, our calculus is capable of performing non-trivial reasoning about rewrite strategies. Specifically, the reasoning about beta-eta normalisation already features strategy combinators, traversals and recursion: the fundamental ingredients of strategic rewriting. As our framework is fully mechanised in Isabelle/HOL, reasoning can be performed directly in and facilitated by the proof assistant. Therefore, it is conceivable — still with a significant effort — to use our framework for reasoning about complex applications, including Elevate [Hagedorn et al. 2020] compiler optimisations. A significant initial hurdle is to encode the language that is rewritten (e.g. the lambda calculus in section 5.3) as well as application-specific rewrites in Isabelle/HOL, before we can start reasoning about the behaviour of more complex rewrite strategies. With our formal calculus and its Isabelle/HOL implementation it would be possible to build up a library of standard languages and rewrites, to facilitate reasoning about increasingly complex practical applications.

6 RELATED WORK

Strategic Rewriting and Traversals. Term rewriting systems [Dershowitz 1985] are a powerful and versatile method to express syntactic transformations. Strategic rewriting languages, which give programmers control over the rewriting process, have seen applications in many areas. Initial efforts, such as the language ELAN [Borovanský et al. 1996], focused on using rewriting as a way to model deduction and computation. The previously mentioned Stratego [Bravenboer et al. 2008; Visser 2001; Visser et al. 1998], which uses System S as its core language, is designed for developing language interpreters in the Spoofax Language Workbench [Wachsmuth et al. 2014]. Elevate [Hagedorn et al. 2020, 2023] is very much in the style of Stratego, but is instead targeted towards guiding optimisations in a compiler for high performance computing. The language TL [Winter and Subramaniam 2004] applies strategic rewriting to data processing tasks, and Strafunski [Lämmel and Visser 2002], which is again a Stratego-like language, uses strategies for datatype-generic programming. Traversals are an essential feature of System S that also appear in other program transformation designs, such as the 'Scrap your boilerplate' (SYB) style traversals (e.g. `everywhere`, `everything`, `anyDescendant`, `anyAncestor` etc.) for XML programming [Lämmel 2007]. Reachability constraints are added to types of these traversals for detecting queries that result in an empty set and transformations that always fail or do not change anything. To analyse strategic programs some algebraic laws are discussed by Cunha and Visser [2007] for equational reasoning and by Lämmel et al. [2013] as hints of potential dead code. One could potentially make use of our weakest precondition calculus to prove and generalise these laws.

Weakest Preconditions. Weakest preconditions were introduced by Dijkstra [Dijkstra 1975]. Bonsangue and Kok [1992] extend Dijkstra's calculus to include recursion in the same way that we do. Weakest preconditions are key to Cook's proof [Cook 1978] of the relative completeness of Floyd-Hoare Logic [Floyd 1967; Hoare 1969], and are similarly used by Goncharov and Schröder [2013] to show relative completeness of their Hoare Logic for programs with monadic effects. Morgan [1994] uses weakest preconditions as the semantic foundation for his refinement calculus, enabling stepwise derivation of programs from their specifications. In recent work, Aguirre et al. [2022] explore the categorical structure of compositional weakest preconditions, characterising them as those that are obtained from the Cartesian lifting of some monad. As a related application of weakest preconditions, Swierstra and Baanen [2019] provide a weakest prediction semantics for effectful programs, accounting for exceptions, state, non-determinism and general recursion. Their work could possibly be an alternative approach to achieve some of the goals of our work, although the application of such a formalism to the form of rewriting in formalisms like system S is not immediate. For example, it is unclear whether System S with its handling of errors and non-termination would actually form a monad. Errors alone can, of course, be handled by the Error

monad; the interaction with divergence and errors is more sophisticated. As a consequence, this may give rise to complications of a similar order of magnitude as the ones addressed in this paper.

Existing Formalisation and Verification. We are not the first to examine strategic rewriting languages formally. Both the initial paper on Stratego [Visser et al. 1998] and the paper on System S [Visser and Benaissa 1998] present big-step operational semantics. However these semantics do not model divergence, and are not the basis for any formal claims. In this work, by contrast, we model all possible outcomes including divergence denotationally, and we show the denotational model equivalent to an extended big-step operational semantics of System S that includes divergence, by establishing the computational soundness and adequacy with respect to the extended big-step operational semantics. Kaiser and Lämmel [2009] formalise a subset of System S without divergence in Isabelle/HOL by shallow embedding, but this formalisation does not include the general fixed-point operator of System S, and the choice to use shallow embedding, while convenient for some tasks, precludes the formalisation of general, meta-theoretic properties about all strategies. In our formalisation, we opt for a deep embedding, enabling us to mechanise all of the definitions and proofs in this paper. Focusing on traversals in strategic languages, Lämmel et al. [2013] characterise a list of strategic programming errors and discuss ways to avoid these errors with static typing and static analysis. With a different approach, we provide a general and formal characterisation of "good" and "bad" strategies as well as successful and unsuccessful executions of strategies, using our location-based weakest precondition calculus.

Kieburtz [2001], an inspiration for this work, informally sketches some weakest precondition rules for Stratego. Rather than a location-based weakest precondition calculus such as ours, Kieburtz [2001] includes assertions in modal logic (specifically a combination of CTL and the modal μ-calculus), where the various tree modalities allow movement to different sub-expressions. However, this modal logic variant does not have the expressive power of our framework because of our choice of location language. For instance, CTL is not expressive enough to reason about the one operator. When it comes to traversals, Kieburtz [2001] does not define general predicate transformers for modal assertions, and thus Kieburtz's [2001] rules do not form a complete calculus. It is not clear how Kieburtz's [2001] approach could be extended to handle traversals in their full generality. In our work, our assertions are just sets of expressions, and we move around an expression by associating locations to our weakest preconditions. This enables us to define general rules for traversals, allowing a compositional and complete calculus for all strategies and all postconditions. In addition, the fixed-point operator is not well constructed in Kieburtz's [2001] work and it is not proven to be monotone, whereas we have a correct treatment of the fixed-point operator and have proven monotonicity of all our formulae. Also, in Kieburtz's [2001] work, soundness is not proven, whereas we prove the soundness of our weakest precondition calculus w.r.t. the formal semantics. Lastly, we provide a careful treatment of divergence with mutually defined wp and $wp^{\uparrow}$, while such a feature is not reflected in Kieburtz's [2001] work.

Type Systems for Strategic Rewriting Languages. A related but parallel strand of work is in giving *types* to strategic rewriting languages. Smits and Visser [2020] add gradual typing to Stratego and use it to find bugs in their strategies for language interpreters. Koppel [2023] uses typed strategies to model multi-language program transformations, Lämmel [2003] adds types to strategies with applications to generic programming in typed languages and Fu et al. [2023] makes use of structural typing with traces for checking ill-composed strategies statically. These type systems emphasise lightweight static or the hybrid of dynamic and static checking to find bugs, whereas our focus is on a complete semantic accounting of rewriting strategies, and the development of a weakest precondition calculus that can demonstrate the absence of bugs, not merely their presence.

Kleene Algebra. Strategic rewriting languages resemble a Kleene Algebra [Kozen 1991] extended with traversals and a biased choice operator. There have been many other extensions to Kleene Algebra, most notably Concurrent Kleene Algebra [Hoare et al. 2011], which adds parallel composition, and Kleene Algebra with Tests [Kozen 1997], which adds Boolean guards to model the semantics of **while** programs. Kozen [1999] shows that reasoning by Kleene Algebra with Tests entirely subsumes Hoare Logic for **while** programs. A version of Kleene Algebra with Tests, NetKAT, has been used to reason about packet switching networks [Anderson et al. 2014]. Recently, Concurrent Kleene Algebra and NetKAT have been combined for reasoning about concurrent network systems [Wagemaker et al. 2022].

Denotational semantics and adequacy. The appeal of the Scott-Strachey approach to semantics [Stoy 1985] is in its local and compositional reasoning, and over the last 50 years it has been used for many diverse programming languages. As far as programming language abstractions go, the strategic rewriting language we consider is mostly standard, and we were able to use the following relevant semantic tools with relatively minor modification. Plotkin pioneered the powerdomain construction [1976] and later characterised it as the free semilattice over a domain [Hennessy and Plotkin 1979]. Most denotational accounts include an adequacy proof, and it is possible to prove them wholesale for standard programming languages with a myriad of expressive features [Johann et al. 2010; Plotkin and Power 2001; Simpson 2004]. We found the decomposition of computational adequacy into dual inductive and coinductive arguments interesting, and we hope it could inform other reflective accounts of adequacy [Devesas Campos and Levy 2018].

7 CONCLUSION AND FUTURE WORK

We have presented Shoggoth, a rigorous formal foundation for strategic rewriting languages, including a comprehensive semantic accounting of System S, and a weakest precondition calculus to facilitate formal reasoning about rewriting strategies. Our semantic treatment models all possible executions of strategies including divergences in both denotational and big-step operational models, and our proofs of soundness and adequacy demonstrate the equivalence of these models. Our location-based weakest precondition calculus is the first formal axiomatic treatment of rewriting strategies, and enables reasoning about traversals by having the notion of location for indicating where in an expression a given strategy operates. Our soundness proof justifies our location-based weakest precondition calculus with respect to our semantic models, and we demonstrate practical application of this calculus by applying it to realistic examples. All of our work has been mechanised in over 5,000 lines of Isabelle/HOL proof script.

Weakest precondition calculi form the basis of *verification condition generators* (VCGs), which are a key component of many automatic and semi-automatic verification tools such as VCC [Cohen et al. 2009] and Dafny [Leino 2010], as well as of static analysers such as the popular Extended Static Checking extension for Java [Flanagan et al. 2002; Leino 2005]. We envision that our weakest precondition calculus could similarly inform the design of a VCG for automatic verification or static checking of rewriting strategies. We intend, in future work, to use Shoggoth as a foundation for the development of tools for verification and, potentially, *synthesis* of rewriting strategies.

ACKNOWLEDGMENTS

Xueying Qin would like to express her special thanks to Professor Glynn Winskel and Professor Dan Ghica for their helpful feedback on the design and equivalence proofs for denotational and operational semantics. Rob van Glabbeek is supported by Royal Society Wolfson Fellowship RSWF\R1\221008. Ohad Kammar is supported by a Royal Society University Research Fellowship. We thank the anonymous reviewers for their insightful suggestions.

DATA AVAILABILITY STATEMENT

The artifact of this paper is on Zenodo [Qin et al. 2023].

REFERENCES

Alejandro Aguirre, Shin-ya Katsumata, and Satoshi Kura. 2022. Weakest preconditions in fibrations. *Math. Struct. Comput. Sci.* 32, 4 (2022), 472–510. https://doi.org/10.1017/S0960129522000330

Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: semantic foundations for networks. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 113–126. https://doi.org/10.1145/2535838.2535862

Marcello M. Bonsangue and Joost N. Kok. 1992. Semantics, Orderings and Recursion in the Weakest Precondition Calculus. In *Sematics: Foundations and Applications, REX Workshop, Beekbergen, The Netherlands, June 1-4, 1992, Proceedings (*LNCS, *Vol. 666)*, J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg (Eds.). Springer, 91–109. https://doi.org/10.1007/3-540-56596-5_30

Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. 1996. ELAN: A logical framework based on computational systems. In *First International Workshop on Rewriting Logic and its Applications, RWLW 1996, Asilomar Conference Center, Pacific Grove, CA, USA, September 3-6, 1996 (Electronic Notes in Theoretical Computer Science, Vol. 4)*, José Meseguer (Ed.). Elsevier, 35–50. https://doi.org/10.1016/S1571-0661(04)00032-5

Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2008. Stratego/XT 0.17. A language and toolset for program transformation. *Sci. Comput. Program.* 72, 1-2 (2008), 52–70. https://doi.org/10.1016/j.scico.2007.11.003

Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (*LNCS, *Vol. 5674)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer, 23–42. https://doi.org/10.1007/978-3-642-03359-9_2

Stephen A. Cook. 1978. Soundness and Completeness of an Axiom System for Program Verification. *SIAM J. Comput.* 7, 1 (1978), 70–90. https://doi.org/10.1137/0207005

Alcino Cunha and Joost Visser. 2007. Transformation of Structure-Shy Programs: Applied to XPath Queries and Strategic Functions. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (Nice, France) *(PEPM '07)*. Association for Computing Machinery, New York, NY, USA, 11–20. https://doi.org/10.1145/1244381.1244385

Nachum Dershowitz. 1985. Computing with Rewrite Systems. *Inf. Control.* 65, 2/3 (1985), 122–157. https://doi.org/10.1016/S0019-9958(85)80003-6

Marco Devesas Campos and Paul Blain Levy. 2018. A Syntactic View of Computational Adequacy. In *Foundations of Software Science and Computation Structures*, Christel Baier and Ugo Dal Lago (Eds.). Springer International Publishing, Cham, 71–87.

Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (1975), 453–457. https://doi.org/10.1145/360933.360975

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended Static Checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, Jens Knoop and Laurie J. Hendren (Eds.). ACM, 234–245. https://doi.org/10.1145/512529.512558

Robert W. Floyd. 1967. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics* 19 (1967), 19–32. https://doi.org/10.1007/978-94-011-1793-7_4

Rongxiao Fu, Ornela Dardha, and Michel Steuwer. 2023. Traced Types for Safe Strategic Rewriting. arXiv:2304.14154 [cs.PL]

Sergey Goncharov and Lutz Schröder. 2013. A Relatively Complete Generic Hoare Logic for Order-Enriched Effects. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*. IEEE Computer Society, 273–282. https://doi.org/10.1109/LICS.2013.33

Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020. Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies. *Proc. ACM Program. Lang.* 4, ICFP (2020), 92:1–92:29. https://doi.org/10.1145/3408974

Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2023. Achieving High Performance the Functional Way: Expressing High-Performance Optimizations as Rewrite Strategies. *Commun. ACM* 66, 3 (2023), 89–97. https://doi.org/10.1145/3580371

Matthew Hennessy and Gordon D. Plotkin. 1979. Full Abstraction for a Simple Parallel Programming Language. In *Mathematical Foundations of Computer Science 1979, Proceedings, 8th Symposium, Olomouc, Czechoslovakia, September 3-7,*

*1979 (*LNCS, *Vol. 74)*, Jirí Becvár (Ed.). Springer, 108–120. https://doi.org/10.1007/3-540-09526-8_8

Tony Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

Tony Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman. 2011. Concurrent Kleene Algebra and its Foundations. *J. Log. Algebraic Methods Program.* 80, 6 (2011), 266–296. https://doi.org/10.1016/j.jlap.2011.04.005

Patricia Johann, Alex Simpson, and Janis Voigtländer. 2010. A Generic Operational Metatheory for Algebraic Effects. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom*. IEEE Computer Society, 209–218. https://doi.org/10.1109/LICS.2010.29

Markus Kaiser and Ralf Lämmel. 2009. An Isabelle/HOL-based model of Stratego-like traversal strategies. In *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal*, António Porto and Francisco Javier López-Fraguas (Eds.). ACM, 93–104. https://doi.org/10.1145/1599410.1599423

Richard B. Kieburtz. 2001. A Logic for Rewriting Strategies. *Electronic Notes in Theoretical Computer Science* 58, 2 (2001), 138–154. https://doi.org/10.1016/S1571-0661(04)00283-X STRATEGIES 2001, 4th International Workshop on Strategies in Automated Deduction - Selected Papers (in connection with IJCAR 2001).

James Koppel. 2023. Typed Multi-Language Strategy Combinators. In *Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands (OASIcs, Vol. 109)*, Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 16:1–16:9. https://doi.org/10.4230/OASIcs.EVCS.2023.16

Dexter Kozen. 1991. A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*. IEEE Computer Society, 214–225. https://doi.org/10.1109/LICS.1991.151646

Dexter Kozen. 1997. Kleene Algebra with Tests. *ACM Trans. Program. Lang. Syst.* 19, 3 (1997), 427–443. https://doi.org/10.1145/256167.256195

Dexter Kozen. 1999. On Hoare Logic and Kleene Algebra with Tests. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*. IEEE Computer Society, 167–172. https://doi.org/10.1109/LICS.1999.782610

Ralf Lämmel. 2003. Typed generic traversal with term rewriting strategies. *J. Log. Algebraic Methods Program.* 54, 1-2 (2003), 1–64. https://doi.org/10.1016/S1567-8326(02)00028-0

Ralf Lämmel. 2007. Scrap Your Boilerplate with XPath-like Combinators. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France) *(POPL '07)*. Association for Computing Machinery, New York, NY, USA, 137–142. https://doi.org/10.1145/1190216.1190240

Ralf Lämmel, Simon Thompson, and Markus Kaiser. 2013. Programming errors in traversal programs over structured data. *Science of Computer Programming* 78, 10 (2013), 1770–1808.

Ralf Lämmel and Joost Visser. 2002. Design patterns for functional strategic programming. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Rule-Based Programming, Pittsburgh, Pennsylvania, USA, 2002*, Bernd Fischer and Eelco Visser (Eds.). ACM, 1–14. https://doi.org/10.1145/570186.570187

K. Rustan M. Leino. 2005. Efficient weakest preconditions. *Inf. Process. Lett.* 93, 6 (2005), 281–288. https://doi.org/10.1016/j.ipl.2004.10.015

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers (*LNCS, *Vol. 6355)*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

Xavier Leroy and Hervé Grall. 2009. Coinductive big-step operational semantics. *Information and Computation* 207, 2 (2009), 284–304. https://doi.org/10.1016/j.ic.2007.12.004 Special issue on Structural Operational Semantics (SOS).

Carroll Morgan. 1994. *Programming from specifications, 2nd Edition.* Prentice Hall.

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. LNCS, Vol. 2283. Springer. https://doi.org/10.1007/3-540-45949-9

Gordon Plotkin and John Power. 2001. Adequacy for Algebraic Effects. In *Foundations of Software Science and Computation Structures*, Furio Honsell and Marino Miculan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–24.

Gordon D. Plotkin. 1976. A Powerdomain Construction. *SIAM J. Comput.* 5, 3 (1976), 452–487. https://doi.org/10.1137/0205035

Xueying Qin, Liam O'Connor, Rob van Glabbeek, Peter Höfner, Ohad Kammar, and Michel Steuwer. 2023. *Artifact for Shoggoth - A Formal Foundation for Strategic Rewriting*. https://doi.org/10.5281/zenodo.10125602

Alex Simpson. 2004. Computational adequacy for recursive types in models of intuitionistic set theory. *Annals of Pure and Applied Logic* 130, 1 (2004), 207–275. https://doi.org/10.1016/j.apal.2003.12.005 Papers presented at the 2002 IEEE Symposium on Logic in Computer Science (LICS).

Jeff Smits and Eelco Visser. 2020. Gradually typing strategies. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020*, Ralf Lämmel, Laurence Tratt, and Juan de Lara (Eds.). ACM, 1–15. https://doi.org/10.1145/3426425.3426928

Matthieu Sozeau. 2014. Proof-relevant rewriting strategies in Coq. In *At Coq Workshop*. https://www.irif.fr/~letouzey/types2014/abstract-13.pdf

J.E. Stoy. 1985. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press. https://books.google.co.uk/books?id=jM0mAAAAMAAJ

Wouter Swierstra and Tim Baanen. 2019. A Predicate Transformer Semantics for Effects (Functional Pearl). *Proc. ACM Program. Lang.* 3, ICFP, Article 103 (jul 2019), 26 pages. https://doi.org/10.1145/3341707

Eelco Visser. 2001. Stratego: A Language for Program Transformation Based on Rewriting Strategies System Description of Stratego 0.5. In *Rewriting Techniques and Applications (*LNCS*, Vol. 2051)*, Aart Middeldorp (Ed.). Springer, 357–361. https://doi.org/10.1007/3-540-45127-7_27

Eelco Visser, Zine El-Abidine Benaissa, and Andrew P. Tolmach. 1998. Building Program Optimizers with Rewriting Strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*, Matthias Felleisen, Paul Hudak, and Christian Queinnec (Eds.). ACM, 13–26. https://doi.org/10.1145/289423.289425

Eelco Visser and Zine El-Abidine Benaissa. 1998. A Core Language for Rewriting. *Electronic Notes in Theoretical Computer Science* 15 (1998), 422–441. https://doi.org/10.1016/S1571-0661(05)80027-1 International Workshop on Rewriting Logic and its Applications.

Guido Wachsmuth, Gabriël D. P. Konat, and Eelco Visser. 2014. Language Design with the Spoofax Language Workbench. *IEEE Softw.* 31, 5 (2014), 35–43. https://doi.org/10.1109/MS.2014.100

Jana Wagemaker, Nate Foster, Tobias Kappé, Dexter Kozen, Jurriaan Rot, and Alexandra Silva. 2022. Concurrent NetKAT - Modeling and analyzing stateful, concurrent networks. In *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings (*LNCS*, Vol. 13240)*, Ilya Sergey (Ed.). Springer, 575–602. https://doi.org/10.1007/978-3-030-99336-8_21

Victor L. Winter and Mahadevan Subramaniam. 2004. The transient combinator, higher-order strategies, and the distributed data problem. *Sci. Comput. Program.* 52 (2004), 165–212. https://doi.org/10.1016/j.scico.2004.03.006

Received 2023-07-11; accepted 2023-11-07