The Journal of Systems and Software 212 (2024) 112006

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

★piler: Compilers in search of compilations☆

Francesco Bertolotti, Walter Cazzola *, Luca Favalli

Università degli Studi di Milano, Computer Science Department, Milan, Italy

ARTICLE INFO

Dataset link: https://doi.org/10.5281/zenodo.8284131

Keywords:
Programming languages
Source-to-source transpilers

ABSTRACT

Compilers pose significant challenges in their development as software products. Language developers face the complexities of ensuring efficiency, adhering to good design practices, and maintaining the overall codebase. These factors make it difficult to predict the unexpected impact of updates on existing software built on the current compiler stack. Furthermore, software created for a specific compiler often lacks reusability for other compiler environments. In this study, we propose a comprehensive framework for the uniform development of compilers that addresses these issues. Our approach involves developing compilers as a collection of small transpilation units, referred to as deltas. The transpilation infrastructure takes source code written in a particular source language and searches for a path of deltas to generate equivalent source code in the target language. By adopting this methodology, language developers can easily update their languages by introducing new deltas into the system. Existing code remains unaffected as old transpilation paths remain available. To support this framework, we have devised a metric space for efficient delta search. This metric space enables us to define a non-overestimating heuristic function, which proves valuable in solving the search problem. Leveraging the A* search algorithm, we can efficiently transpile programs from a source language to the target language. To evaluate the effectiveness of our approach, we conducted a benchmark comparison between the A* search algorithm and the simpler breadth-first search (BFS) algorithm. The benchmark consisted of over 100 transpilation searches, providing valuable insights into the performance and capabilities of this framework.

1. Introduction

Overview. Programming languages are crucial tools for software developers, but building the ecosystem around them, which includes compilers/interpreters, debuggers, and integrated development environments, remains a significant hurdle in programming language development (Bertolotti et al., 2023; Wachsmuth et al., 2014). The research community has put a lot of effort into developing techniques that ease the development of the languages and their ecosystems. Language workbenches (Vacchi and Cazzola, 2015; Kats and Visser, 2010; Klint et al., 2009) and language product lines (Kühn and Cazzola, 2016; Kühn et al., 2019; Méndez-Acuña et al., 2016) are some of the most promising tools for taming the complexity arising from the development of programming languages and their ecosystems. Notwithstanding this, the adoption of new programming languages is struggling to take root. This is because a new language must have an ecosystem as rich as its competitors in order to be accepted, and this can take years of further development (Stefik and Hanenberg, 2014; Meyerovich and Rabkin, 2012, 2013).

Problem statement. To facilitate the widespread adoption of newly introduced programming languages, it is crucial that they provide comprehensive and user-friendly development libraries capable of effectively addressing common tasks across various domains. The software community typically addresses this challenge through various methods, including source-to-source transpilers (Albrecht et al., 1980; Seymour and Dongarra, 2003; Coco et al., 2018), a shared virtual machine (Box and Sells, 2002; Würthinger et al., 2013), inter-process communication (Slee et al., 2007; Vinoski, 1997), and multi-language run-times (Grimmer et al., 2018; Bolz et al., 2009). However, these approaches have some limitations, because even current language workbenches address these issues on a case-by-case basis, an activity that is laborious, demanding, and time-consuming. Unfortunately, traditional language workbenches require a significant amount of effort to seamlessly migrate existing code from one language to another, or even to an updated version of the same language (Bertolotti et al., 2023). Some notable exceptions that offer similar support are Rust and Python code version migration tools.[1,2] This limitation further hinders the adoption and evolution of programming languages.

☆ Editor: Justyna Petke.

This work was partly supported by the MUR project "T-LADIES" (PRIN 2020TL3X8X).

* Corresponding author.
E-mail addresses: bertolotti@di.unimi.it, francesco.bertolotti@unimi.it (F. Bertolotti), cazzola@di.unimi.it (W. Cazzola), favalli@di.unimi.it (L. Favalli).

[1] https://doc.rust-lang.org/edition-guide/editions/transitioning-an-existing-project-to-a-new-edition.html

[2] https://python-modernize.readthedocs.io/en/latest/

https://doi.org/10.1016/j.jss.2024.112006
Received 25 August 2023; Received in revised form 12 January 2024; Accepted 12 February 2024
Available online 1 March 2024

Existing approaches. In the past years, the research community has tackled this problem in a variety of ways. Virtual machines (VMs) usually allow code access from different languages as long as the code can be compiled towards a common (internal) representation. For example, the Scala programming language (Odersky and Rompf, 2014), running on top of the Java virtual machine (Venners, 2000), natively supports interoperability with the Java language ecosystem. Another viable approach is to directly translate the library into a different language. Haxe (Díaz Bilotto and Favre, 2016) programs can be compiled in a variety of languages and it is used to develop the cross-language library. Similarly, the development of language-to-language transpilers (Schultes, 2021) leads to reuse opportunities. Recently, deep learning applications have shown promising results in the area of source-to-source translation (Roziere et al., 2020). Unfortunately, these approaches solve the problem only partially. Using a unique intermediate representation (such as the bytecode) allows to reuse code that can be compiled towards the same bytecode. On the other hand, Haxe still needs to rely on library bindings that support only one of the target languages at a time. Both approaches lack the capabilities to be generalized on scenarios with languages using completely different compilation stacks.

Proposed solution. In this work, we propose a framework—dubbed ★piler[3]—that can automatically compose simple and small transpilation functions—dubbed deltas—to achieve an entire source-to-source translation. These deltas are developed independently from each other. One can achieve complete transpilers by composing deltas. Starting from a program, the subsequent application of deltas induces a graph where nodes are transpiled programs and edges represent the application of a delta (see Fig. 3). By searching on this graph, we can obtain a path of deltas that, composed together, achieves a complete transpilation of the input program. Unfortunately, depending on the topology of the graph, a naive search for the right transpilation can easily become infeasible. However, by establishing a well-defined metric space on the graph, we can equip a search algorithm with a powerful heuristic function that effectively guides the search towards promising directions. A* is a popular (Cui and Shi, 2011) search algorithm that, equipped with a non-overestimating heuristic, performs otherwise long searches in a matter of seconds.

Research questions. To evaluates our framework, we developed three languages: S, S++, and S#. These languages are simplifications of C, C++ and C# respectively. We will use them to answer the following research questions:

RQ_1. Can the ★piler be used to transpile languages in a timely matter?

RQ_2. Can the ★piler be used to migrate a library from a language to another?

RQ_3. What are the strengths and weaknesses the ★piler framework?

In particular, to answer RQ_1, we will empirically evaluate the performance of ★piler on more than 100 translations across different languages with two search algorithms (BFS and A*). Meanwhile, RQ_2 and RQ_3 will be answered from a qualitative perspective.

Organization. The rest of this work is organized as follows. Section 2 introduces notations and known concepts from the literature. Section 3 discusses the ★piler from a theoretical perspective. Section 5 shows a running example of a few simple languages (discussed in Section 4). Section 6 shows the results of the experiments. The results alongside the research question answers are discussed in Section 7. Finally, in Section 8 and Section 9, we discuss related work and draw our conclusions respectively.

[3] ★piler reads as *starpiler* after the A* (Cui and Shi, 2011) search algorithm adopted by the framework.

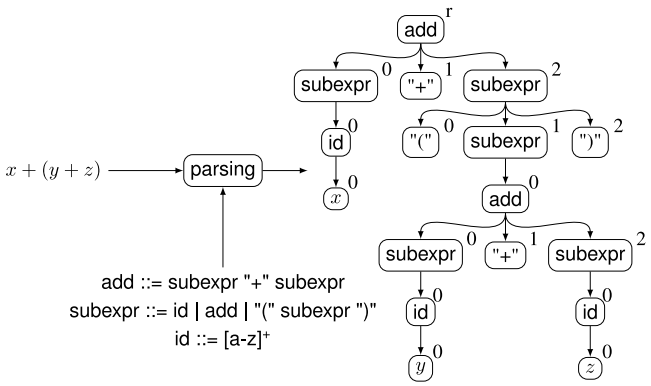


Fig. 1. The parsing process given a simple grammar starting from a text expression to the parse tree.

2. Foundations

In this section, we will briefly introduce concepts and terminologies used in the paper.

Grammar. A grammar is a quadruple $G = (N, \Sigma, R, S)$ where N is a finite set of nonterminal symbols, Σ is a finite set of terminal symbols with $\Sigma \cap N = \emptyset$, R is a finite set of production rules and S is the starting axiom. We use the Backus–Naur form (BNF) to describe the set of rules R. A rule has the form `nonterminal ::= expression`. On the left side of `::=` , there is a symbol from N. On the right side of `::=` there is a sequence of terminal or nonterminal symbols. Here terminal symbols are enclosed with double quotes. To avoid ambiguities, when using multiple grammars, we will denote elements of grammar by adding the grammar name as a subscript to the set name, e.g., Σ_G for the set of terminals of the grammar G.

Concrete Syntax Tree (CST). For our purposes, the CST is a tree representation of a program. The CST is obtained by parsing the program. For example, Fig. 1 shows the CST obtained from parsing the string `" x + ( y + z )"` according to the shown grammar. Formally, a CST is a couple $\tau = (V_G, E_G)$ where $V_G \subseteq N_G \cup \Sigma_G$ and $E_G \subseteq V_G \times V_G$. Finally, τ is connected and acyclic. We use the dot notation $. : V_G \times \mathbb{N} \to V_G$ to denote the sub-tree node of a tree node (numbered from left to right starting from 0). For example, in Fig. 1 we write r.2.1.0 to denote the inner `add` sub-tree. Notice that, Barthwal and Norrish (2009) proved that it is always possible to verify that a CST is generated from a certain grammar. The verification process checks that each node is compliant with the rule that generated the parent. We will write $\tau \lhd G$ when the CST τ is compliant with the grammar G, (i.e., $\tau \in \mathcal{L}_G$ where $\mathcal{L}_G$ is the language defined by G).

Semantics. For our purposes, we use the notation $[\![\tau_G]\!](\rho)$ to denote the evaluation of the CST τ_G on the context ρ (similarly to Tennent (1976)). ρ is a function $\rho : V_G \to \mathcal{X} \cup \{\bot\}$ mapping variable names to values from the semantic domain ($\mathcal{X}$) alongside a special value for undefined variables, $\bot$. The semantic domain is extended with the undefined symbol $\bot$. We will denote with P the set of all possible contexts. For example, the evaluation of the sub-tree r.2.1.0 from Fig. 1 is defined as $[\![r.2.1.0]\!](\rho) = [\![r.2.1.0.0]\!](\rho) + [\![r.2.1.0.2]\!](\rho)$. Instead, $[\![x]\!](\rho) = \rho(x)$ which may have a well-defined value or $\bot$ according to the context function. In the paper, for the sake of brevity, we will rarely explicitly define the semantics of a language, we rather implicitly adopt the intuitive semantics for each construct. For example, the semantics of a `mul` tree is the computation of the multiplication but we will not explicitly define that.

Metric space. A metric space is a couple (X, d) where X is a set of elements and $d : X \to X$ is a non-negative map—called *distance*

function—such that:

1. $\forall x \in X : d(x,x) = 0$.
2. $\forall x, y \in X, x \neq y : d(x,y) > 0$.
3. $\forall x, y \in X : d(x,y) = d(y,x)$.
4. $\forall x, y, z \in X : d(x,z) \leq d(x,y) + d(y,z)$

*A**. The A* algorithm (Hart et al., 1968), a widely-used pathfinding technique, employs a systematic approach to finding the shortest path between two points on a graph or grid while considering associated costs. A* is an informed search algorithm that blends aspects of Dijkstra's algorithm and greedy best-first search. It operates through several key steps: Initializing the open and closed sets, which manage the nodes under evaluation and those already assessed, respectively. The cost and heuristic values for each node are set at the start, with $g(n)$ representing the cost from the start node and $h(n)$ the estimated cost to the goal node, calculated using an admissible heuristic function. The main loop involves selecting nodes with the lowest $f(n)$ values, where $f(n)$ equals the sum of $g(n)$ and $h(n)$. If the goal node is reached, the algorithm ends. Otherwise, it generates successors, calculates tentative g values for neighbors, and updates their attributes accordingly. Subsequently, successors are added to the open set, and the process repeats until the open set is empty or the goal is reached. When the heuristic is admissible, A* is both a complete and optimal pathfinding algorithm, making it invaluable in applications like map routing, video games, robotics, and various other scenarios requiring efficient pathfinding.

3. ★Piler framework

Overview. The ★piler framework is rooted in the concept of delta. Conceptually a delta is a small transpilation unit. As such, a delta is fed with a CST and it returns a CST while preserving the underlying semantics. For example, the identity function is a trivial example of a delta. Also, a full language-to-language transpiler is a proper example of a delta. However, to fully benefit from the framework, deltas should be also compact and reusable units. Thus a correct design of delta should target only a single μ-language, i.e., a small group of language features logically related (Cazzola et al., 2018). For example, a delta could focus only on the transpilation from a for-style loop to a while-style loop. Given a set of these deltas, a target grammar, and a starting program, we want to obtain an equivalent program compliant with the target grammar. Fundamentally, we need to search for a chain of deltas (if one exists) that achieves the desired result (it translates the starting program to the target grammar). The search can be performed by recursively applying deltas to the CST of the starting program. After each application, we check if the current CST is compliant with the target grammar. If it is compliant, we stop. Otherwise, we keep on searching. Trivially, the recursive application of delta can be performed according to BFS. However, this choice, as we will show, renders the ★piler slow. Instead, adopting the A* search algorithms renders the ★piler performant. The usage of A* requires a non-overestimating heuristic function. In this section, we develop a metric space from which the mentioned heuristic is derived.

★Piler foundations. Let us formalize the framework from a theoretical perspective.

Definition 1 (*Node-set*)**.** Let us define the node-set function λ that given a CST $\tau_G = (V_G, E_G)$ returns the set of nonterminals from all non-leaf nodes of the tree, *i.e.,*

$$\lambda(\tau_G) = \left\{ v \mid v \in V_G \wedge \exists w \in V_G \text{ s.t. } (v,w) \in E_G \right\}.$$

For example, the application of the λ function to the root node of the CST in Fig. 1 is $\lambda(r) = \{\mathsf{id}, \mathsf{add}, \mathsf{subexpr}\}$. Notice that the node-set function always returns a subset of the non-terminals of the grammar of the language used to write the program represented by the parse tree.

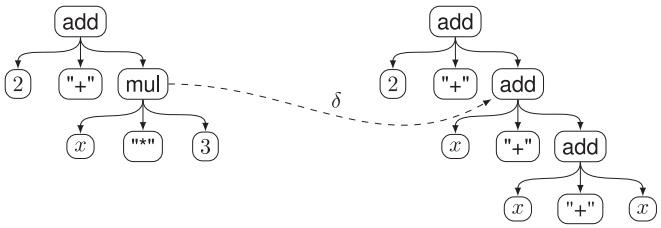


Fig. 2. The application of a delta function—δ—that transpiles the multiplication by a number to sequence of additions.

Definition 2 (*Delta*)**.** Given $G_0, G_1 \in \mathcal{G}$ (set of all possible grammars). A delta is a function, $\delta : \mathcal{T}_{G_0} \to \mathcal{T}_{G_1}$ that maps a CST into another one such that:

$$\forall \rho \in \mathrm{P}, \tau \in \mathcal{T}_{G_0} : [\![\tau]\!](\rho) = [\![\delta(\tau)]\!](\rho) \tag{1}$$

where $\mathcal{T}_G$ is the set of all possible CSTs such that $\tau \in \mathcal{T}_G \implies \tau \lhd G$ (with $G \in \mathcal{G}$).

This means that the application of a delta will not change the result of the evaluation regardless of the program (τ) or context (ρ) at hand. For example, Fig. 2 shows the application of a δ that replaces the mul sub-tree with a sub-tree containing only add nodes. The applied transformation is a *proper delta* as it preserves the semantics of the original CST. Notice that, the grammars G_0 and G_1 may only slightly differ and the difference depends on the syntactic changes applied by the δ function to the original CST. We will often refer to deltas as *transpilation functions*.

It is important to note that while the formal correctness of the ★piler framework depends on the formal correctness of individual deltas, the ★piler can also be used with unverified deltas, similar to other transpiler infrastructures that rely on transpilation passes, such as Keep and Dybvig (2013). Of course, using unverified deltas results in an overall unverified transpiler, which is generally considered acceptable for most applications.

Remark 1 (*Delta Composition*)**.** Given two deltas, δ_1 and δ_2, their composition, $\delta_1 \circ \delta_2$, is still a delta.

This remark can be trivially verified by considering that deltas preserve the CST semantics by definition, therefore

$$\forall \rho \in \mathrm{P}\ [\![\delta_1(\delta_2(\tau))]\!](\rho) = [\![\delta_2(\tau)]\!](\rho) = [\![\tau]\!](\rho).$$

Thus, any finite composition of deltas remains a delta. Potentially, by composing the right deltas, we can achieve full language-to-language transpilation.

Definition 3 (*Transpilation*)**.** Given $\delta_{i_0}, \ldots, \delta_{i_n}$ deltas, we denote their composition $\delta_{i_0} \circ \ldots \circ \delta_{i_n}$ as $\overrightarrow{\delta_I}$, where $I = [i_0, \ldots, i_n]$. Given a CST $\tau_{G_1} \lhd G_1$, $\overrightarrow{\delta_I}$ performs a transpilation to a grammar G_2 if

$$\overrightarrow{\delta_I}(\tau_{G_1}) \lhd G_2$$

Remark 2. If $\overrightarrow{\delta_I}$ is a transpilation from $\tau_1 \lhd G_1$ to $\tau_2 \lhd G_2$ then:

$$\lambda(\overrightarrow{\delta_I}(\tau_1)) \subseteq N_{G_2}$$

This means that we can search for deltas that change the CST node-set to any subset of N_{G_2} and if the resulting tree is a CST belonging to G_2 then, we have obtained a proper transpilation to the target language.

As mentioned earlier, the delta application induces a graph where nodes are CSTs and edges are applications of a delta. For example, Fig. 3 shows a possible graph induced from the application of a few deltas ($\delta_1, \ldots, \delta_6$) to an initial CST τ_1. If τ_6 represents the target CST, to achieve the desired transpilation, we simply need to search for the right sequence of deltas that achieves the desired result. Formally:

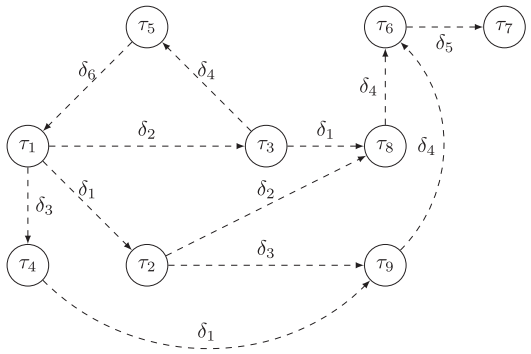


Fig. 3. A graph induced by the application of deltas $\{\delta_1, \delta_2, \delta_3, \delta_4, \delta_5, \delta_6, \}$ on the syntax tree τ_1.

Notation 1 (*Slice*)**.** *We adopt a python-like notation to describe slices of a vector. Let* $x = [x_0, \dots, x_m]$*, then* $x[:k] = [x_0, \dots, x_k]$ *when* $0 \leq k \leq m$*. Otherwise,* $x[:k] = x$*.*

Definition 4 (*Search Graph*)**.** Let $\Delta = \{\delta_0, \dots, \delta_N\}$ be a set of deltas. Let $V \subseteq \mathcal{T}_\Gamma$ be a set of CSTs from grammars $\Gamma = \{G_0, \dots, G_M\}$. Let $E = \{(\tau_1, \tau_2) \in V \times V \mid \exists\ \delta \in \Delta \text{ s.t. } \delta(\tau_1) = \tau_2\}$. We call $S_{\Delta,\Gamma} = (V, E)$ a search graph.

Definition 5 (*Search Problem*)**.** The search problem is defined by the triple $(S_{\Delta,\Gamma} = (V, E), \tau, G)$, where $S_{\Delta,\Gamma}$ is a search graph, $\tau \in V$ is the starting CST, and $G \in \Gamma$ is the target grammar.

Definition 6 (*Search Solution*)**.** Given the search problem $(S_{\Delta,\Gamma} = (V, E), \tau, G)$, a transpilation $\overrightarrow{\delta_I}$ is a solution when:

1. $\overrightarrow{\delta_I}(\tau) \lhd G$.
2. $\forall i \geq 0 : \overrightarrow{\delta_{I[:i]}}(\tau) \in V$
3. $\forall i \geq 0 : (\overrightarrow{\delta_{I[:i]}}(\tau),\ \overrightarrow{\delta_{I[:i+1]}}(\tau)) \in E$

This means that a transpilation is a solution if it maps the input CST to a CST from the target grammar (point 1) while adhering to the search graph definition (points 2 and 3).

The search problem can be solved with algorithms such as breadth-first search (BFS) or depth-first search (DFS). However, since the number of nodes grows exponentially with the number of deltas, the search becomes impractical soon. Notice that, BFS will reach always a solution if one exists. Otherwise, it will explore the entire graph and terminate, if the graph is finite. If the graph is not finite, it may not terminate. To overcome the impracticality issue, we are going to define a simplification of the original search problem.

Definition 7 (*Simplified Search Graph*)**.** Let $\Delta = \{\delta_0, \dots, \delta_N\}$ be a set of deltas. Let $V \subseteq \mathcal{T}_\Gamma$ be a set of CSTs from grammars $\Gamma = \{G_0, \dots, G_M\}$. Let $V' = \{\lambda(\tau) \mid \tau \in V\}$. Let $E' = \{(\lambda(\tau_1), \lambda(\tau_2)) \in V' \times V' \mid \exists\ \delta \in \Delta, \tau_1, \tau_2 \in V \text{ s.t. } \delta(\tau_1) = \tau_2\}$. We call $S'_{\Delta,\Gamma} = (V', E')$ the simplified search graph of the search graph $S_{\Delta,\Gamma} = (V, E)$.

The simplified search graph is simply the search graph where instead of using CSTs as nodes, we use their node-sets. Notice that, the simplified search graph can be built starting from an original search graph.

Definition 8 (*Simplified Search Problem*)**.** The simplified search problem is defined by the triple $(S'_{\Delta,\Gamma} = (V', E'), \tau, G)$, where $S'_{\Delta,\Gamma}$ is a simplified search graph. τ is a starting CST from V'. G is a target grammar from Γ.

Definition 9 (*Simplified Search Solution*)**.** Given the simplified search problem $(S'_{\Delta,\Gamma} = (V', E'), \tau, G)$, a transpilation $\overrightarrow{\delta_I}$ is a solution when:

1. $\lambda(\overrightarrow{\delta_I}(\tau)) \subseteq N_G$.
2. $\forall i \geq 0 : \lambda(\overrightarrow{\delta_{I[:i]}}(\tau)) \in V'$
3. $\forall i \geq 0 : (\lambda(\overrightarrow{\delta_{I[:i]}}(\tau)),\ \lambda(\overrightarrow{\delta_{I[:i+1]}}(\tau))) \in E'$

Theorem 1. *The set of solutions for the search problem is a subset of the solution for the same simplified search problem.*

The proof trivially follows from the fact that a solution for the search problem is always a solution for the simplified search problem. Notice that, the vice versa is not necessarily true. From a practical perspective, by enumerating solutions for the simplified search problem, we will eventually reach a solution for the search problem. Next, we will show that it is possible to build a metric space on the set V' for a simplified search problem. This enables the application of more advanced search algorithms such as A*.

Definition 10 (*Set Difference Distance*)**.** Let $\mathcal{U}$ be a universe set. And let $A, B \subseteq \mathcal{U}$. The function $d_{sdd} : \mathcal{P}(\mathcal{U}) \times \mathcal{P}(\mathcal{U}) \to \mathbb{R}$ (where $\mathcal{P}$ represents the power set function) is defined as:

$$d_{sdd}(A, B) = |A \cup B| - |A \cap B| \tag{2}$$

The set difference function (d_{sdd}) is a proper distance function (Horadam and Nyblom, 2014). It measures the degree to which the input sets differ. For example, if $A = B$ then $d_{sdd}(A, B) = 0$. If $A \cap B = \emptyset$, then $d_{sdd}(A, B) = |A \cup B|$.

Next, we will introduce a novel distance rooted on the set difference distance function.

Definition 11 (*Relative Set Difference Distance*)**.** Let $\mathcal{U}$ be a universe set and $A, B, S \subseteq \mathcal{U}$. The function $d^S_{rsdd} : \mathcal{P}(\mathcal{U}) \times \mathcal{P}(\mathcal{U}) \to \mathbb{R}$

$$d^S_{rsdd}(A, B) = \begin{cases} d_{sdd}(A, B) & \text{if } A \not\subseteq S \wedge B \not\subseteq S, \\ d_{sdd}(S, B) & \text{if } A \subseteq S \wedge B \not\subseteq S, \\ d_{sdd}(A, S) & \text{if } A \not\subseteq S \wedge B \subseteq S, \\ 1/2 & \text{if } A \subseteq S \wedge B \subseteq S \wedge A \neq B, \\ 0 & \text{if } A \subseteq S \wedge B \subseteq S \wedge A = B \end{cases} \tag{3}$$

d_{rsdd} computes the distance between two sets relatively to a third set S: the distance decreases as A and B get closer to S. Fig. 4 shows the behavior of $d^S_{rsdd}(A, S)$ wrt. to the set S. The distance between A and S decreases when the set A approaches to the set S. The distance collapses to the value $1/2$ when the set A is a subset of S but $A \neq S$. Any number $\in (0, 1)$ would comply with the distance function definition: we choose $1/2$ because it is the central number in the range.

Lemma 1. *d^S_{rsdd} is a distance function.*

See Appendix for the formal proof. This lemma allows to induce a metric space on the simplified search graph. In particular, d^S_{rsdd} can be used to build a non-overestimating heuristic required to use the A* search algorithm.

Theorem 2. *Let $\mathcal{N}_\Gamma = \cup_{G \in \Gamma} N_G$ be the set of all nonterminals from grammars in Γ. Let $S \subseteq \mathcal{N}_\Gamma$. Then the couple $(\mathcal{P}(\mathcal{N}_\Gamma), d^S_{rsdd})$ is a metric space.*

The proof trivially follows from Lemma 1. This theorem provides a simple way to build a non-overestimating heuristic required by the search algorithm A*. In particular, the use of A* renders the search for solutions to the simplified search problem particularly fast compared to directly solving the search problem.

Definition 12 (*Heuristic*)**.** Let $h_S : \mathcal{P}(\mathcal{N}_\Gamma) \to \mathbb{R}$ be the positive map

$$h_S(A) = \begin{cases} 0 & \text{if } A \subseteq S \\ d^S_{rsdd}(S, A) & \text{otherwise} \end{cases} \tag{4}$$

where $A, S \subseteq \mathcal{N}_\Gamma$.

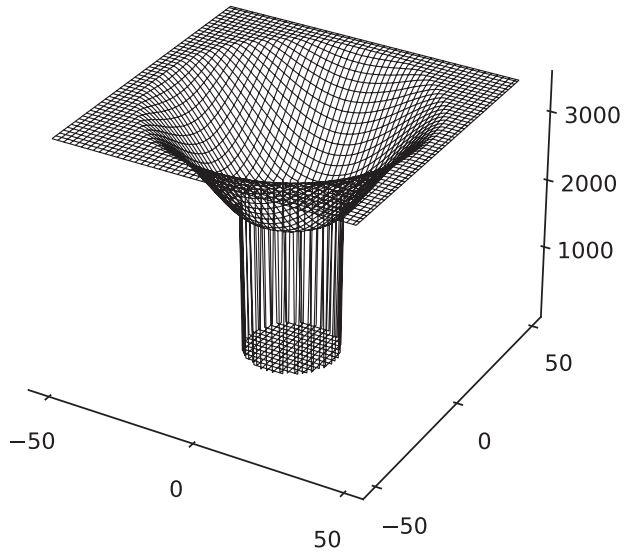


Fig. 4. Distance function d^S_{rsdd} graph. Each point (x, y) represents a set of a ball with center (x, y) and radius 15. The solution set is a ball with center in (0,0) and radius 30. As the set approaches the solution set, the distance decreases. When the set becomes a subset of the solution set, the distance becomes $\frac{1}{2}$.

Theorem 3. *h_S is an admissible heuristic, i.e., non-overestimating.*

Proof. Let $(S'_{\Delta,\Gamma}, \tau, G)$ be a simplified search problem. Where, we search for a transpilation starting from τ to a CST having nonterminals from N_G. Let $S = N_G$. Let τ^* be a CST such that $\lambda(\tau^*) \subseteq N_G$. We need to show that

$$h_S(\lambda(\tau)) \leq d^S_{rsdd}(\lambda(\tau), \lambda(\tau^*))$$

If $\lambda(\tau) \subseteq S$ then:

$$h_S(\lambda(\tau)) = 0 \leq d^S_{rsdd}(\lambda(\tau), \lambda(\tau^*))$$

As d^S_{rsdd} is a distance function, it cannot be negative. Otherwise,

$$h_S(\lambda(\tau)) = d^S_{rsdd}(S, \lambda(\tau)) \tag{5}$$
$$= d_{sdd}(S, \lambda(\tau)) \tag{6}$$
$$\leq d_{sdd}(S, \lambda(\tau)) \tag{7}$$
$$= d_{sdd}(\lambda(\tau), S) \tag{8}$$
$$= d^S_{rsdd}(\lambda(\tau), \lambda(\tau^*)) \quad \square \tag{9}$$

All equations follow from either the definition of function d^S_{rsdd} (Definition 11), or from the definition of the distance function. This theorem shows that it is possible to apply A* to the simplified search problem. For each solution found by A*, we check whether it is also a solution for the original search problem. We stop the search if it is a solution. Otherwise, we continue searching. Algorithm 1 shows the pseudocode for the transpilation. Algorithm 1 is a modified version of the A* algorithm. Notice that, both the search graph and the simplified search graph are never explicitly built. The search graph is explored starting from τ on the fly by applying deltas from Δ.

From the perspective of a ★piler user seeking to transpile language A to language B, the process involves several key steps. The user needs to have a grammar for the target language, a set of deltas capable of translating programs from language A to language B (these deltas can be developed or reused), and a starting CST representing a program from A (possibly obtained by means of parsing). At this point, ★piler takes over, automatically searching for a chain of deltas to transform the starting CST into a CST compliant with the target grammar. It is worth noting that the developed deltas can find utility in novel scenarios requiring transpilation of μ-languages used in languages A and B.

Algorithm 1 A* adapted for the searching a transpilation

Require: Δ set of delta functions
Require: τ input CST
Require: G target grammar
  $C \leftarrow \{\}$
  $O \leftarrow \{\tau\}$
  $g \leftarrow$ map with default value $+\inf$
  $g(\tau) \leftarrow 0$
  $f \leftarrow$ map with default value $+\inf$
  $f(\lambda(\tau)) \leftarrow h_{N_G}(\lambda(\tau))$
  while $O \neq \emptyset$ **do**
    $C \leftarrow o \in O$ with lowest f value
    if $C \subseteq N_G \wedge c \in G$ **then return** c
    end if
    remove c from O.
    for all $\delta \in \Delta$ **do**
      score $\leftarrow g(\lambda(\delta(C))) + d^{N_G}_{rsdd}(\lambda(C), \lambda(\delta(C)))$
      if score $\leq \lambda(\delta(C))$ **then**
        $g(\lambda(\delta(C))) \leftarrow$ score
        $f(\lambda(\delta(C))) \leftarrow$ score $+ h_{N_G}(\lambda(\delta(C)))$
        if $\lambda(\delta(C)) \notin O$ **then**
          $O.add(\lambda(\delta(C)))$
        end if
      end if
    end for
  end while

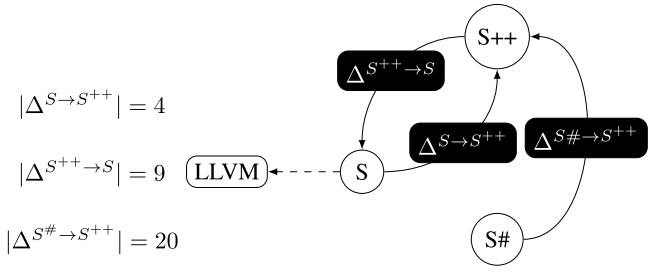


Fig. 5. The system presents three available languages. There are 4 deltas to translate S programs to S++ programs—denoted as $\Delta^{S\to S^{++}}$. There are 9 deltas to translate S++ programs to S programs—denoted as $\Delta^{S^{++}\to S}$. There are 20 deltas to translate S# programs to S++ programs—denoted as $\Delta^{S\#\to S^{++}}$.

4. Languages

We designed three languages to demonstrate the ★piler: S, S++, and S#. The three languages mimic C, C++, and C# respectively. Table 1 summarizes the list of language features for each language. We refer to S as a *sink* language as it can be compiled with tools external (LLVM) to ★piler as shown with the dashed arrow in Fig. 5.

S language. The language S mimics a subset of C. It accounts for expressions, function definitions, and declarations, while loop, if-then, and struct definitions. It supports native types: double, int64, int32, int8, void. For the S language, we also developed a compiler to LLVM (Lattner and Adve, 2004) intermediate representation. S among the developed languages is the only one that can be compiled directly to LLVM. To be executed, other languages (S++ and S#) need to provide either deltas to translate towards another sink language or a direct interpreter/compiler. For example, the following snippet shows the bubble sort algorithm written by using the S language. In S, the left-hand expressions of an assignment are pointers to memory locations where the evaluation of the right-hand expression will be stored. The memory location is accessed by using the & and &[·] operators for variables and arrays respectively.

Table 1
List of language features for S, S++, S# languages. All languages share a set of common features defined in the `Common` entry. The other entries describe the additions wrt. the common features. Notice that even if different languages share the same language feature there may be either syntactic or subtle semantic differences. For example, S++ and S# `new` language features in principle are the same but have different implementations.

Language	Feature
Common	native types: long, int, char, double, float literals: string, array, rationals, naturals import from arithmetic operators: +, -, *, /, % logical operators: ==, !=, >=, <=, <, > other operators: cast to, size of, indexing, enclosed expression, function call statements: if-then, while loop, return, return void, skip, statement expression, assignement, auto assignement, declaration assignement.
S	native types: pointers struct definition function definition global assignement declaration expressions: dereference, reference to
S++	native types: pointers class definition: fields, constructor, destructor, methods function definition global assignement declaration expressions: new, indexing, dereference, reference to
S#	class definition: fields, constructor, methods garbage collector expressions: new, indexing, method call statements: for loop

```
def int64 sort(int64* array, int64 len) does
    int64 i = 0;
    while i < len do
        int64 j = i;
        while j < len do
            if array[i] > array[j] do
                int64 tmp = array[i];
                array&[i] = array[j];
                array&[j] = tmp;
            done
            &j = j + 1;
        done
        &i = i + 1;
    done
    return 0;
done
```

S++ language. The language S++ mimics a subset of C++. It adds support for classes in the S language. As for S, the left-hand expressions in assignments are pointers to memory locations where the evaluation of the right-hand expressions is stored. For classes, the memory location of fields is accessed using the `&.` operator. The `start` method of classes is the constructor of the class. For example, the following snippet shows the definition of an S++ class managing a pair of `int64`.

```
class Pair with
    def int64 a;
    def int64 b;
    def Pair* start(Pair* this, int64 a, int64 b) does
        this&.a = a;
        this&.b = b;
        return this;
    done
end
```

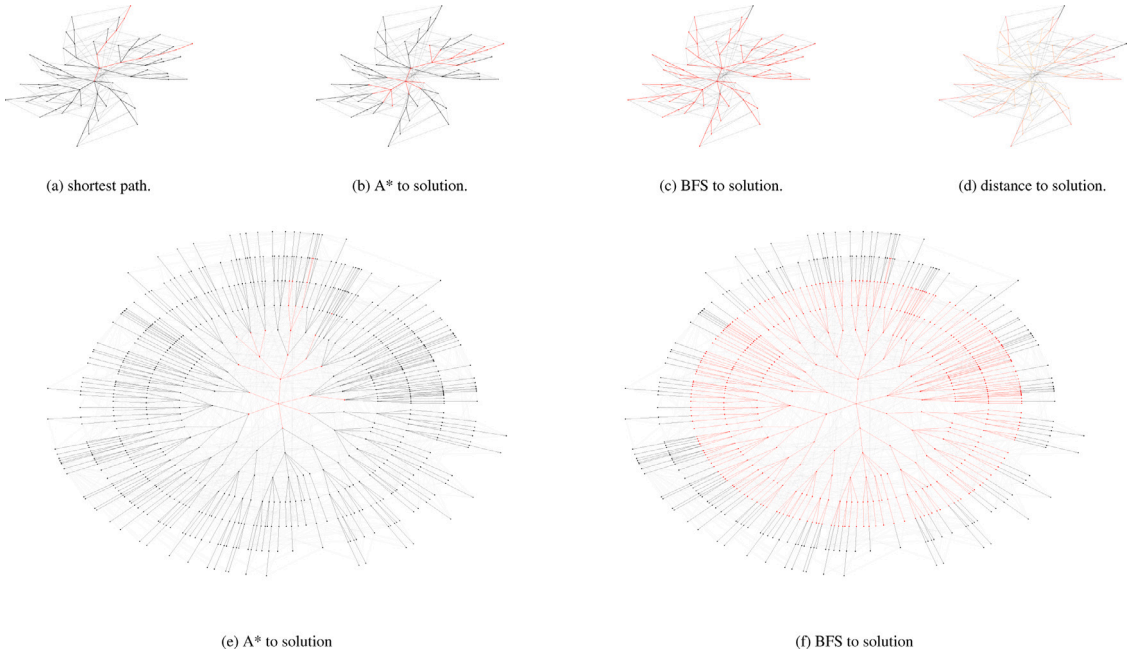
S# language. The language S# mimics a subset of C#. Compared to the S++ language, it removes pointers, it adds a garbage collector that can be explicitly called, and the overall syntax is simplified. Functions are defined by using the `fun` keyword, and function types are defined in advance. The `__init__` method of each class represents its constructor. For example, the following snippet shows an S# class managing a single `int64`. Further, S# adds support for bounded for-loops iterating over a run-time determined number of iterations.

```
class Integer {

    var int64 value;

    fun (Integer -> Integer) __init__ (this) {
        return this;
    }

    fun (Integer, int64 -> Integer) set(this, value) {
        this.value = value;
        return this;
    }

    fun (Integer -> Integer) get(this) {
        return this.value;
    }


}
```

Translations. We developed deltas necessary to transpile S# to S++, S++ to S, and S to S++ to test ★piler performance and capabilities. Fig. 5 schematize the possible translations. We developed 33 deltas to perform these translations. The group of deltas to translate S# to S++ is made of 20 deltas, and it is denoted with $\Delta^{S\#\to S++}$. The group of deltas to translate S++ to S is made of 9 deltas, and it is denoted with $\Delta^{S++\to S}$. The group of deltas to translate S to S++ is made of 4 deltas, and it is denoted with $\Delta^{S\to S++}$. To execute S# programs, it is necessary to translate the program to S++ and then to S which can be executed in the quality of a sink language using the *ad hoc* LLVM compiler. The number of deltas necessary to perform a translation depends on the differences, in terms of language features, between the source and the target languages. For example, S and S++ are fairly similar languages (S++ only adds the classes language feature over S) requiring only a few deltas. On the other hand, S# introduces several language features over S requiring several deltas to get a translation from S to S#. However, note that there are no constraints on the number of deltas, the whole S# to S translation could be performed just with a single deltas. However, a single delta is less likely to be reusable for different languages the best granularity for a delta would consist in translating a single language feature.

5. Running example

Let us consider a simple translation example. The following S# program shows a `Point` class with an addition operation between two points. This class may be part of a bigger library, and we are interested in rendering the same class available to S++ developers.



(a) shortest path. (b) A* to solution. (c) BFS to solution. (d) distance to solution.

(e) A* to solution (f) BFS to solution

Fig. 6. Graphs generated from the exploration of possible translation from a start S# program to the equivalent S++ program. The top row shows the full graph induced using deltas from $\Delta^{S\#\to S++}$. The bottom row shows the graph induced using a larger set of deltas $\Delta^{S\#\to S++} \cup \Delta^{S++\to S}$. Nodes are colored in black. Edges to already explored nodes are colored in light gray. Red colored nodes and edges show the explored paths of different algorithms. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

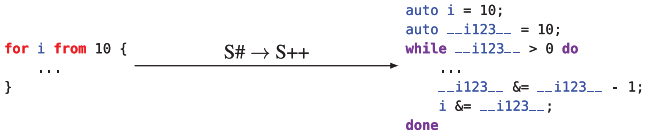


Fig. 7. S# to S++ for-loop delta effect.

```
class Fors(Transformer):
  def ssharplang_for(self, nodes):
    return [
      Tree(Token('RULE', 'spplang_stmt_expr'), [
        Tree(Token('RULE', 'spplang_auto_assignement'), [...]),
        Token('DONE', 'done')]),
      Tree(Token('RULE', 'spplang_while'), [
        Token('WHILE', 'while'),
        ...,
        Token('DO', 'do'),
        Tree(Token('RULE', 'spplang_block'), [...]),
        Token('DONE', 'done')])
    ]
```

Listing 1: S# to S++ for-loop implementation.

```
class Point {
  var double x;
  var double y;

  fun (Point,double,double->Point) __init__(this, x, y) {
    this.x = x;
    this.y = y;
    return this;
  }
  fun (Point,Point->Point) add(this, other) {
    this.x = this.x + other.x;
    this.y = this.y + other.y;
    return this;
  }
}
```

Delta examples. The translation starts from the CST representing the `Point` class. We developed several deltas translating different kinds of S# sub-trees into S++ sub-trees. For example, we developed a delta that translates S# style for-loops into S++ while loops (Fig. 7). The delta introduces an unused identifier `__i123__` used to iterate over the while. At the end of each iteration, the original variable name `i` is reassigned to the value of `__i123__` to simulate the for-loop. Listing shows a portion of the delta implementation using a Lark Transformer.[4] An application of this delta replaces all sub-trees representing the S# for-loop and attaches the corresponding while loop in S++. Similarly, we implemented the delta that deals with S# style assignments. Since in S++, the left-hand expression of an assignment needs to be a memory location, we need to translate the field access operator (dot) into the operator that returns memory locations (reference dot). Therefore, the snippet `this.x = x;` becomes `this&.x = x;` . Another delta deals with the function definition sub-tree, and it rearranges the types to fit the S++ function definition style. The snippet

```
fun (Point,Point->Point)add(this,other){...}
```

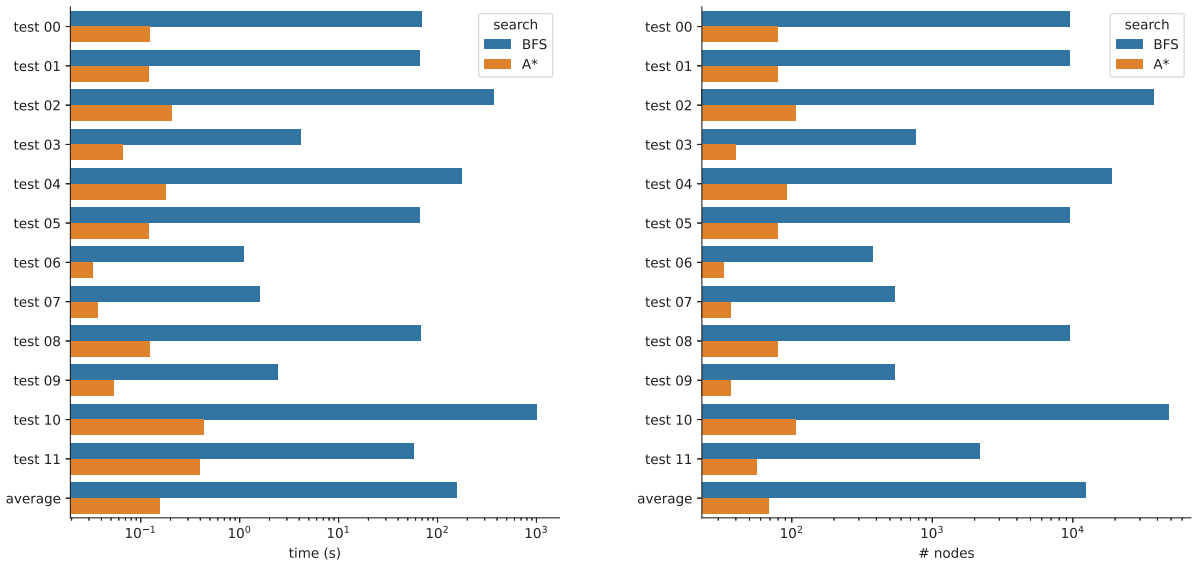
becomes

```
def Point* add(Point* this, Point* other)does ...done.
```

We are left with the CST representing the transpilation (provided that the CST respects the S++ grammar (Barthwal and Norrish, 2009)) when all the S# language features have been exhausted. The following snippet shows the full translation.

```
class Point with
  def double x;
  def double y;

  def Point* start(Point* this, double x, double y) does
    this&.x = x;
    this&.y = y;
    return this;
  done
  def Point* add(Point* this, Point* other) does
```

[4] https://lark-parser.readthedocs.io/en/latest/visitors.html#lark.visitors.Transformer



(a) Test times.

(b) node visited.

Fig. 8. Test times and node visited for each test in the $S\# \to S{+}{+}$ translation task.

```
    this&.x = this.x + other.x;
    this&.y = this.y + other.y;
    return this;
  done
end
```

Induced graph. A delta application may depend on the application of previous deltas. For example, if we were to transpile the same `Point` class into the S language, we would need to transpile S# language features to the respective intermediate S++ language feature, as we did not develop any delta from S# to S (see Fig. 5). Of course, the path of deltas to perform the translation may not be known beforehand. Therefore, we need a search step to find the proper transpilation. As mentioned earlier, the application of deltas induces a graph on which we search for a solution. Consider Fig. 6, sub-figures from 6(a) to 6(d) shows the graph induced for translating the S# class `Point` into the S++ class `Point` using only the group $\Delta^{S\#\to S{+}{+}}$. Fig. 6(a) shows, in red, the shortest path from the root node to a solution. Fig. 6(b) shows, in red, the nodes explored by the A* algorithm. Fig. 6(c) shows, in red, the nodes explored using a BFS algorithm. Fig. 6(d) shows, edges and nodes colored according to the distance from the solution wrt. the distance function d^{S}_{rsdd}. All these graphs, in light gray, show which edges connect a node to another explored node. Most noticeably, the A* algorithm explores a fraction of the nodes explored by the BFS algorithm. Now, let us consider Figs. 6(e) and 6(f). These figures show the induced graph when using $\Delta^{S\#\to S{+}{+}} \cup \Delta^{S{+}{+}\to S}$. Most noticeably, the A* algorithm explores a number of nodes that is roughly the same as the previous case. Instead, the BFS algorithm explores a larger set of nodes, thus requiring more time.

6. Evaluation

In this section, we will evaluate the compilation of several programs with respect to a BFS and A* searches.

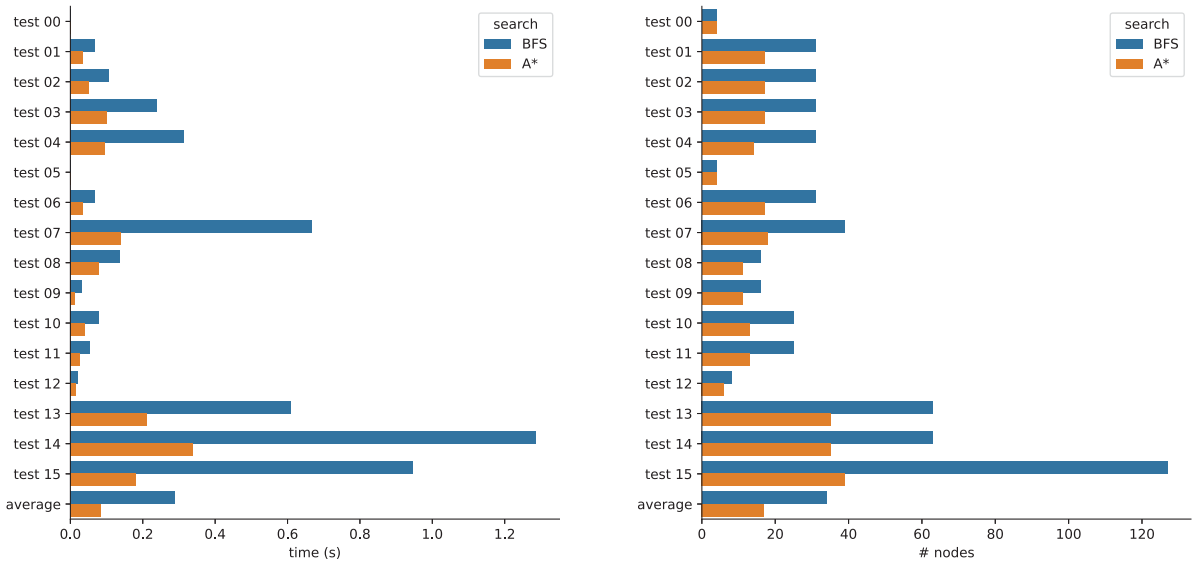
Setup. The evaluation is performed on a PC with 32 GB of available memory and processor Intel Core i7-10700K.

Data. We develop 12 tests for the $S\# \to S{+}{+}$ translation task, 16 tests for the $S{+}{+} \to S$ translation task, and 92 tests for the $S \to S{+}{+}$ translation task. The number of tests for translate $S \to S{+}{+}$ is higher as we reused the test for the LLVM compiler. Each test is run for 5 times (for a total of 620 runs). All tests are manually handcrafted to span all language features of the respective languages.

Results S# to S++. This translation task usually induces a large graph with thousands of nodes, as the number of deltas usable for the translation is 20. The main results are highlighted in Fig. 8. Fig. 8(a) shows the transpilation times for all the 16 tests of the $S\# \to S{+}{+}$ translation task. The time axis is displayed on a logarithmic scale. Using the A* search algorithm, to execute all the tests requires less than 1 s. On average, the completion time of all tests is 0.16 s. Instead, the tests (to translate S# to S++) may require several minutes to complete, ranging from 1 s to 20 min when using BFS. On average, the completion time is 158.52 s. Fig. 8(b) shows the number of nodes visited for each test. On average, A* visits 68 nodes (ranging from 33 to 106) and BFS visits 12,265 nodes (ranging from 375 to 48,117). Thanks to the discussed heuristic in Section 3, A* is capable of exploring far fewer nodes compared to BFS, resulting in small translation times.

Results S++ to S. This translation task induces a far smaller graph with hundreds of nodes, as the number of deltas usable for translation is only 9. The main results are highlighted in Fig. 9. Fig. 9(a) shows the transpilation times for all 12 tests of the $S{+}{+} \to S$ translation task. The time axis is displayed with a linear scale. Using the A* search algorithm, the average completion time is 0.09 s, ranging from 1 ms to 0.3 s. Instead, using BFS, the tests run for 0.29 s on average, ranging from 1 ms to 1.29s. Fig. 9(b) shows the number of nodes visited for each test. On average, A* visits 16 nodes (ranging from 4 to 39) and BFS visits 34 nodes (ranging from 4 to 127). Again, the A* algorithm shows noticeable improvement over the BFS algorithm.

Results S to S++. This translation task induces a very small graph with less than 10 nodes, as the number of deltas usable for translation is only 4. The main results are highlighted in Fig. 10. Fig. 10(a) shows the transpilation times for all the 92 tests of the $S \to S{+}{+}$ translation task. The time axis is displayed with a linear scale. Using the A*



(a) Test times.

(b) node visited.

Fig. 9. Test times and node visited for each test in the $S{+}{+} \rightarrow S$ translation task.

search algorithm, the average completion time is 0.966 ms. Using the BFS algorithm, the average completion time is 8.156 ms. Fig. 10(b) shows the number of nodes visited for each test. On average, the A* algorithm visits 2.60 nodes, ranging from 2 to 7 nodes. Instead, the BFS algorithm visits 2.63 nodes on average, ranging from 2 to 8 nodes. In this scenario, the A* algorithm does not lead to meaningful improvements. Instead, due to the higher initialization costs, it leads to slightly lower performance.

7. Discussion

Compilation times. We can now answer to:

RQ$_1$. Can ★piler be used to transpile languages in a timely matter?

When using search algorithms without a heuristic (such as BFS), the compilation times render the ★piler impractical. However, when using A* with the proposed heuristic h_S, the compilation times improve drastically. In our experiments, all tests require less than 1 second to complete the transpilation. Section 6 suggests that ★piler could be applied in more complex scenarios without becoming impractical due to compilation times.

The transpilation time is heavily influenced by the size of the search graph and the number of nodes that must be traversed to reach a solution. In general, the smaller the search graph, the less time transpilation is expected to require, irrespective of the search algorithm employed. The most impactful way to keep the search graph small is by limiting the number of deltas ($|\Delta|$). When using BFS, having a small number of deltas is crucial to achieve transpilation times on the order of seconds. Meanwhile, in all our experiments, the designed heuristic successfully guided A*'s search in the right direction, ensuring small transpilation times regardless of the search graph's dimensions.

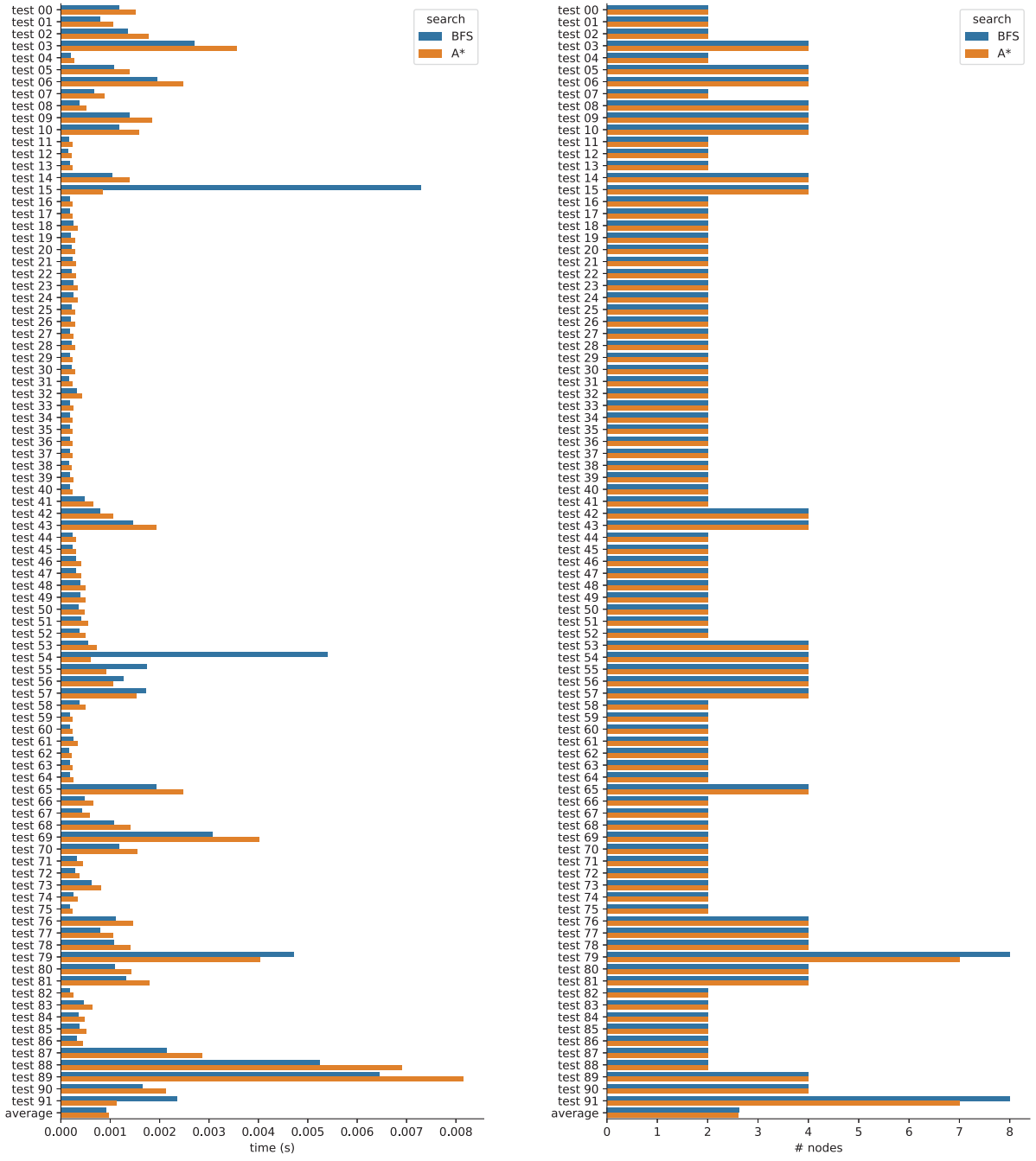
Libraries migration. Often, developed libraries are specific for the language in which they are developed. Reusing these pieces of software becomes extremely difficult when different languages and different VMs are involved. However, by performing a translation, the same software developed in S++ can be reused with the S language. For example, we developed a small class managing strings in S++. The class uses stdio[5] directives to efficiently compute string operations. By translating the S++ class also S developers can access a higher-level API to handle strings. However, in the general case, migration through transpilation requires a fair amount of carefulness. For example, in Python class methods with specific names (such as `__init__` or `__getitem__`) can have a specific behavior. Instead, these names add no special meaning for Java or C++. Therefore, if a Java method named `__getitem__` were to be transpiled in Python, another name would have to be generated, and generating a meaningful new name may be difficult. However, when using the ★piler framework developers can define the deltas that most fit the developer needs case by case. For example, a developer may choose to use a delta that generates a new method name randomly, or he could use a delta that prompts for the new name when one is needed. Developers could use deltas that generate new method names using a deep learning architecture as Bertolotti and Cazzola (2023b,a) and Alon et al. (2019). Another difficult example is the Python `eval` and `exec` which allows to executes strings containing Python code. Since these strings cannot be inferred at compile time, when transpiling from Python to Java, the target language will need the ability (natively or with an external library) to execute Python strings. These cases can be handled in the ★piler framework by writing a delta that translates the `exec` in a library call that can execute Python code strings. This is just to mention a couple of all the possible tricky language feature-dependent translations.

We can now answer to:

RQ$_2$. Can ★piler be used to migrate a library from a language to another?

The ★piler allows for translation of libraries from a source language to a target language. However, when speaking of real-world language, developers are likely to incur in difficult-to-handle scenarios. Some instances may have multiple ways to be handled, and others may require a large amount of work. Developers using ★piler can handle multiple choices using a different set of deltas to perform a transpilation. Also, using ★piler, developers have the choice to develop only a core set of deltas that handle core language features instead of dealing with all the possible language features. In this scenario, ★piler will perform translations using only the available deltas, meaning that in some cases it will not reach a solution.

[5] https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/stdio.h.html



(a) Test times. (b) node visited.

Fig. 10. Test times and node visited for each test in the $S{+}{+} \rightarrow S$ translation task.

Reusing compiler tests. Compiler tests are one of the most valuable assets to verify the correctness of compilers. The research community has developed several tools to efficiently generate test suites for several compilers (Chen et al., 2021; Sun et al., 2016). However, generated suites are specific to the target language. Using the proposed framework, a test suite can be translated and reused to test new languages. A single suite may be capable of testing different languages. For example, in our demonstration experiment, test programs developed for S++ and S# are also used to test the S language. Also, many of the S test programs are translated to S++ to test the language.

Reusing compiler components. Introducing new features in existing languages can be difficult (Nystrom et al., 2003). Instead, with ★piler the language developer needs only to add a syntactic construct and develop a delta performing the translation of the new feature. For example, in our demonstration experiment, we introduced the **for loop** language feature in the S# language then we added a delta function to translate the S# for loop to the S++ *while loop*. The same translation can be reused in other languages that implement the S# style for loop. For example, if we were to introduce S# style for loop to S++, we would need to only add the syntactic construct to the language and the system will automatically take care of the application of the respective delta to reach a translation.

Heuristic. The proposed heuristic h_S is effective in exploiting the language features inside a syntax tree to guide a translation system. However, there are cases when this heuristic is of little help. Recall that, the heuristic starts to lower when, during search, the current syntax tree

shares language features with the solution set. Otherwise, the value of the heuristic does not shrink. For example, consider Fig. 4, on the edges of the graph the distance to the solution remains the same, thus not guiding the search towards the solution. In these cases, applying A* is equivalent to applying a BFS search. This means that A* cannot help on those cases that require a chain of deltas that does not get close to the solution set early in the search. However, it is still possible to guide the search, but it will require extra knowledge. For example, if we know that translation needs to go through a certain set of language features S', we can craft another heuristic to guide the specific case:

$$h_{S,S'}(A) = \begin{cases} (0,0) & \text{if } A \subseteq S \\ (d^{S}_{rsdd}(S,A), d^{S'}_{rsdd}(S',A)) & \text{otherwise} \end{cases} \tag{10}$$

Among nodes that have the same distance to the solution set S, it promotes those that are close to the intermediate set S'. Of course, the heuristic can be extended with several intermediate sets to guide even delta paths that are expected to be extremely long.

Deltas development. Deltas are functions developed in isolation from each other. This yields a system that can reuse already developed components easily. Yet during searches, deltas can interact with each other unexpectedly as unforeseen situations appear. Further, the correctness of a delta is, in many cases, impractical to verify. Thus, the development of delta should also include extensive unit and integration tests to identify potential issues and bugs.

Deltas debugging. Consider a transpilation path $\vec{\delta}$ that ends in an error (or an unexpected result). Most likely, this is due to a bug in one of the deltas used in $\vec{\delta}$. Yet, tracking the error is difficult, as it could have occurred at any point in the chain. Moreover, if we consider that these functions may be working on a very large CST, pinpointing the bug may be extremely difficult and time-consuming. Therefore, if deltas are not properly organized, it may result in a brittle system. Works such as Shi et al. (2020) could also be used to mitigate these issues.

Deltas as compilers. We discussed a system that searches among a dataset of deltas for a path to a correct transpilation. However, developers may design deltas so that a specific application chain works for every input program, *i.e.,* a compiler.

Further properties on deltas. In this work, we assumed that deltas preserve the semantics of the translated programs. However, one could design deltas to satisfy other properties (e.g., security properties (Abate et al., 2019)). The requirement that such properties would need to satisfy is compositionally (i.e., the composition of deltas satisfy the desired property must satisfy the same property). This consideration may also open up also possible variation of the current framework, that given deltas satisfying different properties, try to find a compilation path that maximizes the number of satisfied properties.

Different programming paradigms. Developing a transpiler becomes increasingly challenging when dealing with different programming paradigms. Unfortunately, we do not anticipate that ★piler would significantly reduce the inherent complexity of such tasks. However, ★piler can be a valuable tool for enhancing reusability. As long as a 'bridge' of deltas exists for transpiling between different paradigms, developers can direct their attention to transpiling between similar paradigms. The ★piler will automatically compose deltas to bridge across paradigm shifts without requiring explicit specification.

You can choose your transpilation. Recall that the language S++ denotes the constructor method using the identifier `start`. If we were to translate the S# class that already contained a method named `start`, we would run quickly into a dilemma. We cannot transpile the S# `start` method as is, because it would be regarded as a constructor in the target language. Yet, changing its signature may be a problem as the S++ API would change. These cases may or may not require an *ad hoc* treatment. To solve these cases, you can choose to use the delta that handles methods with the name `start` or not. This feature does make for customizable transpilers that can handle a variety of cases depending on the developer's needs.

Partial transpilers. Transpiling a program in one language into another is not always possible. For example, transpiling a program written in a Turing complete language into a non-Turing complete language is not always possible. However, some programs may still be transpilable. Consider a program that performs only assignments and basic operations without relying on loops. Such a program can be transpiled into a non-Turing complete language (as long as it supports assignments). In these cases, the system will use only the deltas to translate assignments. If an unlimited loop is present in the source program, the transpilation will fail, as there will be no delta available to translate the loop into the target language.

We can now answer to:

RQ_3 What are the strengths and weaknesses the ★piler framework?

★piler framework presents a different way to develop compilers and transpilers. The reuse of libraries, compiler tests, and deltas themselves are the main compelling advantages of ★piler. Moreover, ★piler is a flexible framework, that can work with any subset of the deltas necessary to translate all programs from a source language to a target one. On the other hand, the development of deltas is the main challenge of the framework. As mentioned before, deltas can be difficult to debug as they may interact with each other in unpredicted ways.

7.1. Threats to validity

External validity. In this work, we assert that the ★piler framework represents a valid alternative for developing programming languages. However, it is important to note that the framework has only been tested with small programming languages and small programs. To validate our claims, we would need to scale up both the languages and tests to ensure its applicability in all practical scenarios.

Internal validity. As we observed, using A* in comparison to BFS results in acceptable transpilation times. However, the compilation/transpilation time within the ★piler framework depends on various factors, including computer hardware and the topology of the search graph. To substantiate our claims, we conducted numerous tests on languages (S, S++, and S#) that encompass a reasonable variety of features, all of which were performed on the same hardware.

Construct validity. In our efforts of answering RQ_1, we aimed to measure the timeliness of ★piler. In order to do so, we gauged the execution times. However, these measurements can be subject to a variety of perturbations difficult to control. To mitigate these issues, we provide measurements from an average of 5 runs for each test. Furthermore, we introduce a hardware-independent metric, the number of explored nodes, which is independent of hardware and software perturbations.

8. Related works

Language workbenches. While we rarely discuss language workbenches, the proposed system can be regarded as a language workbench. One language workbench that we already mentioned is Neverlang (Vacchi and Cazzola, 2015). Another language workbench risen in popularity is Spoofax (Wachsmuth et al., 2014; Kats et al., 2010). Spoofax uses the Stratego to define the language semantics as a chain of transformations. A different approach is taken by CBS (Churchill et al., 2015; Mosses, 2019b,a). CBS is based on the concept of funcons, a semantic unit that can be reused. The PlanCompS (van Binsbergen et al., 2016) projects aim to collect a variety of funcons to cover all the language developers needs. Instead, MPS (Völter and Pech, 2012; Pech et al., 2013) is a non-textual language workbench in which the AST is directly written without the aid of a parser. Another example of a language workbench

is MontiCore (Grönninger et al., 2008; Krahn et al., 2010; Rumpe et al., 2021). MontiCore uses a specific domain-specific language to define a grammar. Instead, the semantics are implemented in Java through visitors. The Truffle framework (Wimmer and Würthinger, 2012) uses tree transformation to optimize code during runtime. Usually, language workbenches offer an infrastructure to develop compilers and language tools using syntactic and semantic rules. For example, the Neverlang language workbench offers a development infrastructure to develop languages using semantic rules that run on the Java virtual machine. Instead, the ★piler does not define semantic rules explicitly but it defines a translation between semantically equivalent CSTs. In comparison to other language workbenches, ★piler offers the advantage of allowing the definition of programming languages based on previously implemented ones with minimal or no glue code. This is made possible as ★piler transpilations are automatically searched without requiring the explicit definition of the transpilation path. Furthermore, ★piler enables the automatic definition of transpilations to all reachable languages in the search graph. However, it is important to note that ★piler is not currently as mature as other more established language workbenches and lacks the support or development environments that make these workbenches desirable.

Transpilers. Transpilers are software that aims to translate one program written in one language into a different one. For example, SequalsK (Schultes, 2021) is a bidirectional transpiler between Swift and Kotlin. SequalsK aims to bridge the development of Android and IOS applications. Ling et al. (2022) have developed a C to Rust source-to-source transpiler with the purpose of migrating old source code. Also, a partial Python to Rust transpiler is proposed by Lunnikivi et al. (2020). They have shown a 12x performance improvement in the transpiled code. These transpilers are only concerned with a single source language and a single target language (one-to-one). Instead, the ★piler framework offers a homogeneous approach to developing many-to-many transpilers. ROSE (Quinlan, 2000; Quinlan and Liao, 2011) is a compiler infrastructure with an intermediate representation that supports a variety of languages, such as C, C++, and Java. ROSE infrastructure uses tree transformation to apply optimizations on the intermediate representation. EpsilonFlock (Rose et al., 2010, 2014) is a tool developed using the Epsilon platform (Kolovos, 2008) to perform a rule-based meta-model transpilation. Instead, of using an intermediate representation to which all languages need to be transpiled, the ★piler framework allows a level of flexibility that permits the development of both one-to-one transpilers and one-to-many transpilers. Moreover, delta developed for any transpiler can always be reused for different scenarios. Nanopass (Sarkar et al., 2004) utilizes transpilation units, referred to as 'passes' instead of 'deltas,' to define comprehensive transpilers and compilers. While it provides a more sophisticated definition of passes, Nanopass lacks the search step introduced by ★piler. A significantly distinct approach to compiler design can be seen in program synthesis, as demonstrated in Bhatia et al. (2023). Although promising, program synthesis is currently mainly applied to code fragments rather than full programs, in order to keep the search space manageable.

Multi-language systems. Folliot et al. (1998) describe a multi-language system—called VVM. VVM is a virtual machine running VMlets that can execute the bytecode of their language. Self (Wolczko et al., 1999) is a minimal programming language that is optimized during runtime. Self has been used to implement languages such as Java and Smalltalk without relying on custom VMs. VMs usually run a single intermediate language. However, they can be effective at unifying the ecosystem of the languages that compile towards a VM (successful examples are Java and Scala programming languages). Instead, the ★piler framework can be effective also at unifying the ecosystem of languages that are developed to run on different VMs. Moreover, the ★piler framework can decouple a language from its VM. It can allow to run the same language on different VMs.

9. Conclusion

In this work, we designed a system to build transpilers. The system uses a collection of user-provided CST transformation—called deltas—to search for a desired transpilation. Deltas are small transpilation functions that can be reused across different transpilations or during the same transpilation. The system induces a graph generated by the recursive application of deltas. The system searches for a transpilation to the desired target language in the induced graph. Provided that deltas are all correct if the system terminates, it reaches a correct transpilation. The transpilation search can be informed with a heuristic to reach a solution in a timely fashion compared to a BFS search. We prove that such a heuristic exists, and it is admissible. Finally, we compare the search algorithm BFS and A* on a benchmark of more than 100 cases. We show that the A* algorithm, equipped with the discussed heuristic, performs extremely better compared to the alternative. You can find the code for the ★piler alongside the experiment data at:

https://doi.org/10.5281/zenodo.8284131

CRediT authorship contribution statement

Francesco Bertolotti: Writing – review & editing, Writing – original draft, Software, Methodology, Investigation, Conceptualization. **Walter Cazzola:** Writing – review & editing, Writing – original draft, Supervision, Project administration, Methodology, Investigation, Funding acquisition, Conceptualization. **Luca Favalli:** Software, Methodology, Investigation, Conceptualization.

Declaration of competing interest

We wish to confirm that there are not known conflicts of interest associated with this publication and there has been no significant financial support for this work that could have influenced its outcome.

Data availability

A replication package with the obtained results is available on Zenodo: https://doi.org/10.5281/zenodo.8284131.

Appendix. Metric space proof

Let $\mathcal{U}$ be the universe set. Let $\mathcal{X} \subseteq \mathcal{P}(\mathcal{U})$. Let $S \in \mathcal{X}$. We will show that $(\mathcal{X}, d^S_{rsdd})$ is a proper metric space. To do so, we need to show that d^S_{rsdd} is a distance function. Let us use $\overline{d}(\cdot,\cdot)$ instead of d^S_{rsdd} and $\underline{d}$ instead of d_{sdd}.

1. $A \in \mathcal{X} \implies \overline{d}(A,A) = 0$.
2. $A, B \in \mathcal{X} \wedge A \neq B \implies \overline{d}(A,B) > 0$.
3. $A, B \in \mathcal{X} \implies \overline{d}(A,B) = \overline{d}(B,A)$.
4. $A, B, C \in \mathcal{X} \implies \overline{d}(A,C) \leq \overline{d}(A,B) + \overline{d}(B,C)$.

The first three properties immediately follow from the definition of $\overline{d}$. The last property (triangle inequality) will be proved by exhaustively checking all cases.

Case $A = B = C$:

$$A,B,C \subseteq S: \quad \underbrace{\overline{d}(A,C)}_{0} \leq \underbrace{\overline{d}(A,B)}_{0} + \underbrace{\overline{d}(B,C)}_{0}$$

$$A,B,C \nsubseteq S: \quad \underbrace{\overline{d}(A,C)}_{\underline{d}(A,C)} \leq \underbrace{\overline{d}(A,B)}_{\underline{d}(A,B)} + \underbrace{\overline{d}(B,C)}_{\underline{d}(B,C)}$$

Case $A = B \neq C$:

$$\overline{d}(A,C) \leq \underbrace{\overline{d}(A,B)}_{0} + \overline{d}(B,C)$$

Case $A \neq B = C$:

$$\overline{d}(A,C) \leq \overline{d}(A,B) + \underbrace{\overline{d}(B,C)}_{0}$$

Case $A = C \neq B$:

$$\underbrace{\overline{d}(A,C)}_{0} \leq \overline{d}(A,B) + \overline{d}(B,C)$$

Case $A \neq C \neq B$:

$$\begin{aligned}
A,B,C \subseteq S &: && \underbrace{\overline{d}(A,C)}_{1/2} \leq \underbrace{\overline{d}(A,B)}_{1/2} + \underbrace{\overline{d}(B,C)}_{1/2} \\
A \subseteq S \wedge B,C \not\subseteq S &: && \underbrace{\overline{d}(A,C)}_{\underline{d}(S,C)} \leq \underbrace{\overline{d}(A,B)}_{\underline{d}(S,B)} + \underbrace{\overline{d}(B,C)}_{\underline{d}(B,C)} \\
B \subseteq S \wedge A,C \not\subseteq S &: && \underbrace{\overline{d}(A,C)}_{\underline{d}(A,C)} \leq \underbrace{\overline{d}(A,B)}_{\underline{d}(A,S)} + \underbrace{\overline{d}(B,C)}_{\underline{d}(S,C)} \\
C \subseteq S \wedge A,B \not\subseteq S &: && \underbrace{\overline{d}(A,C)}_{\underline{d}(A,S)} \leq \underbrace{\overline{d}(A,B)}_{\underline{d}(A,B)} + \underbrace{\overline{d}(B,C)}_{\underline{d}(B,S)} \\
A,B \subseteq S \wedge C \not\subseteq S &: && \underbrace{\overline{d}(A,C)}_{\underline{d}(S,C)} \leq \underbrace{\overline{d}(A,B)}_{1/2} + \underbrace{\overline{d}(B,C)}_{\underline{d}(S,C)} \\
A,C \subseteq S \wedge B \not\subseteq S &: && \underbrace{\overline{d}(A,C)}_{1/2} \leq \underbrace{\overline{d}(A,B)}_{\underline{d}(S,B)} + \underbrace{\overline{d}(B,C)}_{\underline{d}(B,S)} \\
B,C \subseteq S \wedge A \not\subseteq S &: && \underbrace{\overline{d}(A,C)}_{\underline{d}(A,S)} \leq \underbrace{\overline{d}(A,B)}_{\underline{d}(A,S)} + \underbrace{\overline{d}(B,C)}_{1/2} \\
A,B,C \not\subseteq S &: && \underbrace{\overline{d}(A,C)}_{\underline{d}(A,C)} \leq \underbrace{\overline{d}(A,B)}_{\underline{d}(A,B)} + \underbrace{\overline{d}(B,C)}_{\underline{d}(B,C)}
\end{aligned}$$

This concludes the proof. $\overline{d}(\cdot,\cdot)$ is a distance function. Thus, $(\mathcal{X}, \overline{d})$ is a metric space.

References

Abate, C., Blanco, R., Grag, D., Hriţcu, C., Patrignani, M., Thibault, J., 2019. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. In: Delaune, S., Jia, L. (Eds.), Proceedings of the 32nd Computer Security Foundations Symposium. CSF'19, IEEE, Hoboken, NJ, USA, pp. 256–271.

Albrecht, P.F., Garrison, Philip E., Graham, S.L., Hyerle, R.H., Ip, P., Krieg-Brückner, B., 1980. Source-to-source translation: ada to pascal and pascal to ada. ACM Sigplan Not. 15, 183–193.

Alon, U., Brody, S., Levy, O., Yahav, E., 2019. code2seq: Generating sequences from structured representations of code. In: Rush, A. (Ed.), Proceedings of the 7th International Conference on Learning Representations. ICLR'19, New Orleans, LA, USA.

Barthwal, A., Norrish, M., 2009. Verified, executable parsing. In: Castagna, G. (Ed.), Proceedings of the 18th European Symposium on Programming. ESOP'09, Springer, York, United Kingdom, pp. 160–174.

Bertolotti, F., Cazzola, W., 2023a. CombTransformers: statement-wise transformers for statement-wise representations. IEEE Trans. Softw. Eng. 49, 4677–4690. http://dx.doi.org/10.1109/TSE.2023.3310793.

Bertolotti, F., Cazzola, W., 2023b. Fold2Vec: towards a statement based representation of code for code comprehension. Trans. Softw. Eng. Methodol. 32, 6:1–6:31. http://dx.doi.org/10.1145/3514232.

Bertolotti, F., Cazzola, W., Favalli, L., 2023. On the granularity of linguistic reuse. J. Syst. Softw. 202, http://dx.doi.org/10.1016/j.jss.2023.111704.

Bhatia, S., Kohli, S., Seshia, S.A., Cheung, A., 2023. Building code transpilers for domain-specific languages using program synthesis. In: Ali, K., Salvaneschi, G. (Eds.), Proceedings of the 37th European Conference on Object-Oriented Programming. ECOOP'23, Schloss Dagstuhl - Leibniz Center für Informatik, Seattle, WA, USA, pp. 38:1–38:30.

Bolz, C.F., Cuni, A., Fijalkowski, M., Rigo, A., 2009. Tracing the meta-level: PyPy's tracing JIT compiler. In: Rogers, I. (Ed.), Proceedings of the 4th Worksop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems. ICOOOLPS'09, ACM, Genoa, Italy, pp. 18–25.

Box, D., Sells, C., 2002. Essential.Net: The Common Language Runtime. Vol. 1, Addison-Wesley.

Cazzola, W., Chitchyan, R., Rashid, A., Shaqiri, A., 2018. μ-DSU: A micro-language based approach to dynamic software updating. Comput. Languag. Syst. Struct. 51, 71–89. http://dx.doi.org/10.1016/j.cl.2017.07.003.

Chen, J., Patra, J., Pradel, M., Xiong, Y., Zhang, H., Hao, D., Zhang, L., 2021. A survey of compiler testing. ACM Comput. Surv. 53, 4:1–4:36.

Churchill, M., Mosses, P.D., Schulthorpe, N., Torrini, P., 2015. Reusable components of semantic specifications. Trans. Aspect-Oriented Softw. Dev. 12, 132–179.

Coco, E.J., Osman, H.A., Osman, N.I., 2018. JPT: a simple java-python translator. Comput. Appl.: Int. J. 5.

Cui, X., Shi, H., 2011. A*-based pathfinding in modern computer games. Int. J. Comput. Sci. Netw. Secur. 11, 125–130.

Díaz Bilotto, P.N., Favre, L., 2016. Migrating java to mobile platforms through HaXe: an MDD approach. In: Rosado da Cruz, A.M., Paiva, S. (Eds.), Modern Software Engineering Methodologies for Mobile and Cloud Environments. IGI Global, pp. 240–268, (chapter 13).

Folliot, B., Piumarta, I., Riccardi, F., 1998. A dynamically configurable, multi-language execution platform. In: Guedes, P., Bacon, J. (Eds.), Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications. EW'98, ACM, Sintra, Portugal, pp. 175–181.

Grimmer, M., Schatz, R., Seaton, C., Würthinger, T., Luján, M., Mössenböck, H., 2018. Cross-language interoperability in a multi-language runtime. Trans. Progr. Lang. Syst. 40, 8:1–8:43. http://dx.doi.org/10.1145/3201898.

Grönninger, H., Krahn, H., Rumpe, B., Schindler, M., Völkel, S., 2008. MontiCore: a framework for the development of textual domain specific languages. In: Schäfer, W., Dwyer, M., Gruhn, V. (Eds.), Companion Proceedings of the 30th International Conference on Software Enginering. Companion ICSE'08, IEEE, Leipzig, Germany, pp. 925–926.

Hart, P.E., Nilsson, N.J., Raphael, B., 1968. A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Syst. Sci. Cybern. 4, 100–107.

Horadam, K., Nyblom, M., 2014. Distances between sets based on set commonality. Discrete Appl. Math. 167, 310–314.

Kats, L.C.L., Visser, E., 2010. The spoofax language workbench: rules for declarative specification of languages and IDEs. In: Rinard, M., Sullivan, K.J., Steinberg, D.H. (Eds.), Proceedings of the ACM International Conference on Object-Oriented Programming Systems Languages and Applications. OOPSLA'10, ACM, Reno, Nevada, USA, pp. 444–463.

Kats, L.C.L., Visser, E., Wachsmuth, G., 2010. Pure and declarative syntax definition: paradise lost and regained. In: Proceedings of ACM Conference on New Ideas in Programming and Reflections on Software. Onward! 2010, ACM, Reno-Tahoe, Nevada, USA.

Keep, A.W., Dybvig, R.K., 2013. A nanopass framework for commercial compiler development. In: Uustalu, T. (Ed.), Proceedings of the 18th International Conference on Functional Programming. ICFP'13, ACM, Boston, MA, USA, pp. 343–350.

Klint, P., van der Storm, T., Vinju, J., 2009. EASY meta-programming with rascal. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (Eds.), Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering III. GTTSE'09, Springer, Braga, Portugal, pp. 222–289.

Kolovos, D., 2008. An Extensible Platform for Specification of Integrated Languages for Model Management (Ph.D. thesis). University of York, York, United Kingdom.

Krahn, H., Rumpe, B., Völkel, S., 2010. MontiCore: a framework for compositional development of domain specific languages. Int. J. Softw. Tools Technol. Transf. 12, 353–372.

Kühn, T., Cazzola, W., 2016. Apples and oranges: comparing top-down and bottom-up language product lines. In: Rabiser, R., Xie, B. (Eds.), Proceedings of the 20th International Software Product Line Conference. SPLC'16, ACM, Beijing, China, pp. 50–59.

Kühn, T., Cazzola, W., Giampietro, N.Pirritano., Poggi, M., 2019. Piggyback IDE support for language product lines. In: Thüm, T., Duchien, L. (Eds.), Proceedings of the 23rd International Software Product Line Conference. SPLC'19, ACM, Paris, France, pp. 131–142.

Lattner, C., Adve, V., 2004. LLVM: a compilation framework for lifelong program analysis and transformation. In: Smith, M.D. (Ed.), Proceedings of the 2nd International Symposium on Code Generation and Optimization. CGO'04, IEEE, San José, CA, USA, pp. 75–86.

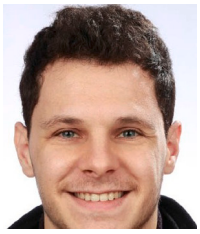
Ling, M., Yu, Y., Wu, H., Wang, Y., Cordy, J.R., Hassan, A.E., 2022. In rust we trust: a transpiler from unsafe c to safer rust. In: Companion Proceedings of the 44th International Conference on Software Engineering. ICSE'22-Companion, IEEE, Pittsburgh, PA, USA, pp. 354–355.

Lunnikivi, H., Jylkkä, K., Hämäläinen, T., 2020. Transpiling python to rust for optimized performance. In: Orailoglu, A., Jung, M., Reichenbach, M. (Eds.), Proceedings of the 20th International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation. SAMOS'20, Springer, Samos, Greece, pp. 127–138.

Méndez-Acuña, D., Galindo, J.A., Degueule, T., Combemale, B., Baudry, B., 2016. Leveraging software product lines engineering in the development of external DSLs: a systematic literature review. Comput. Lang. Syst. Struct. 46, 206–235.

Meyerovich, L.A., Rabkin, A.S., 2012. Socio-PLT: principles for programming language adoption. In: Edwards, J. (Ed.), Proceedings of the ACM international Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. Onward!'12, ACM, Tucson, AZ, USA, pp. 39–54.

Meyerovich, L.A., Rabkin, A.S., 2013. Empirical analysis of programming language adoption. In: Hosking, A.L., Eugster, P., Videira-Lopes, C. (Eds.), Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. OOPSLA'13, ACM, Indiana, IN, USA, pp. 1–18.
Mosses, P.D., 2019a. A component-based formal language workbench. In: Monahan, R., Prevosto, V., Proença, J. (Eds.), Proceedings of the 5th Workshop on Formal Integrated Development Environment. F-IDE'19, Porto, Portugal, pp. 29–34.
Mosses, P.D., 2019b. Software meta-language engineering and CBS. J. Comput. Lang. 50, 39–48.
Nystrom, N., Clarkson, M.R., Myers, A.C., 2003. Polyglot: an extensible compiler framework for java. In: Proceedings of the 12th International Conference on Compiler Construction. CC'03, Springer, Warsaw, Poland, pp. 138–152.
Odersky, M., Rompf, T., 2014. Unifying functional and object-oriented programming with scala. Commun. ACM 57, 76–86.
Pech, V., Shatalin, A., Völter, M., 2013. JetBrains MPS as a tool for extending java. In: Binder, W. (Ed.), Proceedings of the 10th International Conference on Principles and Practices of Programming on the Java Platform. PPPJ'13, ACM, Stuttgart, Germany, pp. 165–168.
Quinlan, D., 2000. ROSE: compiler support for object-oriented frameworks. Parallel Process. Lett. 10, 215–226.
Quinlan, D., Liao, C., 2011. The rose source-to-source compiler infrastructure. In: Midkiff, S., Eigenmann, R., Bae, H. (Eds.), Proceedings of the Cetus Users and Compiler Infrastructure Workshop. Galveston, TX, USA, pp. 1–3.
Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.C., 2010. Model migration with epsilon flock. In: Tratt, L., Gogolla, M. (Eds.), Proceedings of the 3rd International Conference on Theory and Practice of Model Transformations. ICMT'10, Springer, Malága, Spain, pp. 184–198.
Rose, L.M., Kolovos, D., Paige, R.F., Polack, F.A.C., Poulding, S., 2014. Epsilon flock: a model migration language. Softw. Syst. Model. 13, 735–755.
Roziere, B., Marie-Anne, L., Chanussot, L., Lample, G., 2020. Unsupervised translation of programming languages. In: Ranzato, M., Hadsell, R., Balcan, M.F., Lin, H.T. (Eds.), Proceedings of the 34th Conference on Neural Information Processing Systems. NeurUPS'20, Curran Associates Inc, pp. 20601–20611.
Rumpe, B., Hölldobler, K., Kautz, O., 2021. MontiCore: language workbench and library. In: Handbook. Aachen, Germany.
Sarkar, D., Waddell, O., Dybvig, R.K., 2004. A nanopass infrastructure for compiler education. In: Okosaki, C., Fisher, K. (Eds.), Proceedings of the 9th International Conference on Functional Programming. ICFP'04, ACM, Snow Bird, UT, USA, pp. 201–212.
Schultes, D., 2021. SequalsK—a bidirectional swift-kotlin-transpiler. In: Abreu, R., Fazzini, M. (Eds.), Proceedings of the 8th International Conference on Mobile Software Engineering and Systems. MobileSoft'21, IEEE, Madrid, Spain, pp. 73–83.
Seymour, K., Dongarra, J., 2003. Automatic translation of fortran to JVM bytecode. Concurr. Comput.: Pract. Exper. 15, 207–222.
Shi, K., Lu, Y., Chang, J., Weu, Z., 2020. PathPair2Vec: an AST path pair-based code representation method for defect prediction. J. Comput. Lang. 59.
Slee, M., Agarwal, A., Kwiatkowski, M., 2007. Thrift: scalable cross-language services implementation. White Paper 5. Facebook.
Stefik, A., Hanenberg, S., 2014. The programming language wars: questions and responsibilities for the programming language community. In: Krishnamurthi, S. (Ed.), Proceedings of the ACM international Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. Onward!'14, ACM, Portland, OR, USA, pp. 283–299.
Sun, C., Le, V., Su, Z., 2016. Finding compiler bugs via live code mutation. In: Smaragdakis, Y. (Ed.), Proceedings of the 31st International Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA'16, ACM, Amsterdam, Netherlands, pp. 849–863.
Tennent, R.D., 1976. The denotational semantics of programming languages. Commun. ACM 19, 437–453.
Vacchi, E., Cazzola, W., 2015. Neverlang: a framework for feature-oriented language development. Comput. Languag. Syst. Struct. 43, 1–40. http://dx.doi.org/10.1016/j.cl.2015.02.001.
van Binsbergen, L.T., Sculthorpe, N., Mosses, P.D., 2016. Tool support for component-based semantics. In: Companion Proceedings of the 15th International Conference on Modularity. Companion Modularity'16, ACM, Málaga, Spain, pp. 8–11.
Venners, B., 2000. Inside the Java Virtual Machine. McGraw-Hill.
Vinoski, S., 1997. CORBA: integrating diverse applications within distributed heterogeneous environments. IEEE Commun. Mag. 35, 46–55.
Völter, M., Pech, V., 2012. Language modularity with the MPS language workbench. In: Proceedings of the 34th International Conference on Software Engineering. ICSE'12, IEEE, Zürich, Switzerland, pp. 1449–1450.
Wachsmuth, G.H., Konat, G.D.P., Visser, E., 2014. Language design with the spoofax language workbench. IEEE Softw. 31, 35–43.
Wimmer, C., Würthinger, T., 2012. Truffle: a self-optimizing runtime system. In: Leavens, G.T. (Ed.), Proceedings of the 3rd Annual Conference on Systems, Programming and Applications: Software for Humanity. SPLASH'12, ACM, Tucson, AZ, USA, pp. 1–2.
Wolczko, M., Ole, A., Ungar, D., 1999. Towards a universal implementation substrate for object-oriented languages. In: Proceedings of the OOPSLA Workshop on Simplicity, Performance and Portability in Virtual Machine Design. WSPPVMD'99, Denver, CO, USA.
Würthinger, T., Wimmer, C., Woß, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D., Wolczko, M., 2013. One VM to rule them all. In: Hirschfeld, R. (Ed.), Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software. Onward!'13, ACM, Indianapolis, IN, USA, pp. 187–204.



Francesco Bertolotti is currently a Computer Science Ph.D. student at Università degli Studi di Milano and a member of the ADAPT Laboratory. Previously, he was an assistant researcher at the same University where he also got his master degree in Computer Science. His research interests are programming languages, software quality, machine/deep learning techniques and their reciprocal cross-fertilization.

Walter Cazzola is currently a Full Professor in the Department of Computer Science of the Università degli Studi di Milano, Italy and the Chair of the ADAPT laboratory. Dr. Cazzola designed the mChaRM framework, @Java, [a]C#, Blueprint programming languages and he is currently involved in the designing and development of the Neverlang language workbench. He also designed the JavAdaptor dynamic software updating framework and its front-end FiGA. He has written over 100 scientific papers. His research interests include (but are not limited to) software maintenance, evolution and comprehension, programming methodologies and languages. He served on the program committees or editorial boards of the most important conferences and journals about his research topics. He is associate editor for the Journal of Computer Languages published by Elsevier. More information about Dr. Cazzola and all his publications are available at http://cazzola.di.unimi.it.

Luca Favalli is currently a Computer Science Postdoctoral Researcher at Università degli Studi di Milano. He got his Ph.D. in computer science from the Università degli Studi di Milano. He is involved in the research activity of the ADAPT Lab and in the development of the Neverlang language workbench and of JavAdaptor. His main research interests are software design, software (and language) product lines and dynamic software updating with a focus on how they can be used to ease the learning of programming languages.