One Tool, Many Languages: Language-Parametric Transformation with Incremental Parametric Syntax

JAMES KOPPEL, MIT, USA
VAROT PREMTOON, MIT, USA
ARMANDO SOLAR-LEZAMA, MIT, USA

We present a new approach for building source-to-source transformations that can run on multiple programming languages, based on a new way of representing programs called **incremental parametric syntax**. We implement this approach in Haskell in our Cubix system, and construct incremental parametric syntaxes for C, Java, JavaScript, Lua, and Python. We demonstrate a whole-program refactoring tool that runs on all of them, along with three smaller transformations that each run on several. Our evaluation shows that (1) once a transformation is written, little work is required to configure it for a new language (2) transformations built this way output readable code which preserve the structure of the original, according to participants in our human study, and (3) our transformations can still handle language corner-cases, as validated on compiler test suites.

CCS Concepts: • **Software and its engineering → Translator writing systems and compiler generators**; *Syntax*; **General programming languages**;

Additional Key Words and Phrases: program transformation, refactoring, expression problem

ACM Reference Format:
James Koppel, Varot Premtoon, and Armando Solar-Lezama. 2018. One Tool, Many Languages: Language-Parametric Transformation with Incremental Parametric Syntax. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 122 (November 2018), 28 pages. https://doi.org/10.1145/3276492

1 INTRODUCTION

In 2014, Dropbox had a massive refactoring to do. They wanted to let users log in with both a personal and corporate account on the same computer, but they had built the client assuming users only had one account. To change this, they needed to pass along information about which account each operation was for, and thread an Account parameter through tens of thousands of functions. Many program transformation experts could have readily built a tool for this, though it would have been a quite expensive task for one use-case, and Dropbox opted not to hire one. And so, in a company of over 100 engineers, the top project of the year was to tediously add parameters to functions.

Back in 2010, Facebook had a similar problem. All sorts of privacy bugs were being exposed by the media, like weird combinations of settings that would let someone view another user's private photos. Facebook assembled a crack team; they needed this problem fixed quickly, and made to never return. The privacy checks were too haphazard: a bunch of conditionals every place where photos may be displayed. They needed to move these all to one place: private photos would never be fetched from the database in the first place. To do this, they needed to add a ViewerContext

Authors' addresses: James Koppel, MIT, Cambridge, MA, USA, jkoppel@mit.edu; Varot Premtoon, MIT, Cambridge, MA, USA, varot@mit.edu; Armando Solar-Lezama, MIT, Cambridge, MA, USA, asolar@csail.mit.edu.

2475-1421/2018/11-ART122
https://doi.org/10.1145/3276492

parameter to tens of thousands of functions. And so, for several weeks, every waking moment of several dozen of their top engineers was spent adding parameters to functions.

One might think that a clairvoyant entrepreneur in 2010 could have built a tool for this problem, and sold it to both Facebook and Dropbox. Alas, no, for Facebook's codebase was in PHP, while Dropbox's was in Python. And, with today's methods, building a similar program transformation tool for different languages requires **building it separately for each language**.

We are not the first to notice the *language-parametric transformation* problem of building a single transformation that can run on multiple languages. Intuitively, this should be possible: languages have a lot of similarities, and humans can readily apply the same refactoring in many different languages. The challenge then is to find some way to capture the similarities across languages, while being flexible enough to express their differences.

The obvious approach is to convert many languages into a single *intermediate representation*. Unfortunately, doing so inevitably loses information. While this is fine for code-generation or analysis, it fails for source-to-source transformations, which must produce an output similar to the input. Instead, IR-based tools are known to "mutilate" the program, such as by converting all for-loops into while-loops.

There is another line of work that promises the kind of flexible representations needed: instead of building one representation to represent all languages, having a different representation for each language, but letting them share common fragments. This is the approach taken by previous work on modular syntax [Bahr and Hvitved 2011; Zhang et al. 2015], along with its cousin work on modular interpreters [Liang et al. 1995] and modular semantics [Delaware et al. 2013; Mosses 2004]. In principle, these techniques could be used to do language-parametric transformation, but the previous work does not scale to real languages. All these approaches assume that the entire language is built from these generic fragments. Hence, one would have to do huge amounts of up-front work to define fragments capable of representing all variations of each feature of modern programming languages, and assemble them into representations for each language. The difficulty of developing language-parametric infrastructure has meant that previous work in this space, such as the work funded by the Dutch program on language-parametric program restructuring [Heering and Lämmel 2004; Lämmel 2002], has all been for DSLs, toy languages, and language subsets.

This paper presents the first work that builds source-to-source transformations that run on multiple real languages. Our key insight is a new representation called **incremental parametric syntax** (IPS). In incremental parametric syntax, languages are represented using a mixture of language-specific and generic parts. Like previous work on modular syntax, transformations deal only with the generic fragments. Unlike previous work, the implementer starts with a pre-existing normal syntax definition, and only does enough up-front work to redefine a small fraction of a language in terms of these generic fragments. Rather, they can *incrementally* convert more of a language to generic fragments, as needed by new transformations. Best of all, since IPSs are defined as a "diff" to an existing syntax definition, implementations can re-use third-party language frontends.

We have implemented incremental parametric syntax in a Haskell framework called Cubix, and implemented support for 5 languages: C, Java, JavaScript, Lua, and Python. To evaluate Cubix, we built several program transformations that each run on multiple of those languages. We show transformations built in this style can have readable output, unlike IR-based approaches: our "Turing test" human study shows their output is no less readable than hand-transformed code. We show transformations built in this style can handle language corner-cases: the example transformations pass 100% of compiler test suites, excluding some self-referential tests that should not pass ("assert function foo is declared on line 37") and tests that break the third-party parsers and pretty-printers. Finally, using Cubix, we created a prototype tool for threading variables throughout chains of

function calls, as in Dropbox and Facebook's problem, and implemented it for all 5 language simultaneously (including Python, but not yet PHP).

1.1 Why IRs Don't Solve Multi-Language Transformation

An old idea for building multi-language tools is to translate each language into some *intermediate representation*. This works for writing analyses and code-generators, but is a poor fit for source-to-source transformation, which must preserve information.

Conceptually, the IR approach to analysis is to provide a family of `lower` functions of type `C → IR`, `Java → IR`, etc, which transform each language into the IR, along with an `analyze` function of type `IR → AnalysisResult`. Similarly, the IR approach to code generation provides a term of type `IR`, and `lift` functions of type `IR → C`, `IR → Java`, `IR → Python`, etc. The natural extension to transformation is to implement a `transform` function of type `IR → IR`, and compose it with the `lower` and `lift` functions to get language-specific transforms of type `Java → Java`, `C → C`, etc. But this makes a promise which is too good to be true: one can compose the `lower` and `lift` functions to get a translation from any language to any other!

The catch is that tools that implement this approach "mutilate" the program. Most commonly, the IR will be some kind of least common denominator of the supported languages, seen in frameworks like SAIL [Dillig et al. 2009] and BAP [Brumley et al. 2011], and bytecodes such as LLVM [Lattner and Adve 2004] and the JVM [Lindholm et al. 2014]. If the IR only supports `while`-loops, then any transformation through this `IR` will convert all loops into `while`-loops, even if the transformation has nothing to do with loops. Information about the original program has been lost. The alternative is for the IR to be a union of all concepts of the languages. The Clang AST, for instance, contains separate node types for both Objective-C and C++ exception-handling. This approach essentially still requires the user to write a transformation separately for each language: it can use the same node to represent similar constructs in different languages *only if* they are exactly identical. And, among its many other drawbacks, it still loses information about what's *not* in the program (e.g.: Java contains no pointer arithmetic, which simplifies analysis).

The end result is: because of these problems with conventional approaches, at time of writing, we are aware of no previous framework that allows the user to define a single program transformation, run it on programs from multiple languages, and obtain output suitable for humans.

1.2 Incremental Parametric Syntax

So, one-size-fits-all IRs don't work. Our solution is to find a way to apply parametric polymorphism to program transformations. The high-level idea of *incremental parametric syntax* is to build transformations with the following functions:

$$
\begin{aligned}
&\mathsf{decompose}_J :: \mathsf{Java} \to \mathsf{Generic} \bowtie \mathsf{Remainder}_J\\
&\mathsf{decompose}_C :: \mathsf{C} \to \mathsf{Generic} \bowtie \mathsf{Remainder}_C\\
&\mathsf{transform} :: \forall x.\ \mathsf{Generic} \bowtie x \to \mathsf{Generic} \bowtie x\\
&\mathsf{recompose}_J :: \mathsf{Generic} \bowtie \mathsf{Remainder}_J \to \mathsf{Java}\\
&\mathsf{recompose}_C :: \mathsf{Generic} \bowtie \mathsf{Remainder}_C \to \mathsf{C}
\end{aligned}
$$

Here, languages are decomposed into generic and language-specific parts. Then a transformation can be run on the generic parts, while preserving the rest of the program so that high-quality source code may be reconstructed. Unlike the common IR approach, these type signatures guarantee that a transformation cannot modify the language-specific parts, and the `decompose` and `recompose` functions cannot be used to translate one language into another. And rather than construct the generic/language-specific decomposition up-front, IPS allows a programmer to begin with a third-party frontend for each language, and incrementally shift pieces of the language into the generic fragment as needed for new transformations. Hence, developers can add support for a new language

in less than two days of work — much of this time is spent looking at the language spec to understand how to model it in terms of generic components.

The composition $X \bowtie Y$ is done using an approach known in term-rewriting as "sum of signatures" and known in the functional-programming community as "data types à la carte" [Swierstra 2008]. This approach can modularly define node types and mix-and-match them between languages, but does not let these nodes differ between languages: it cannot use the same notion of variable declarations to model both C declarations (which have types) and JavaScript ones (which do not). Similarly, in this approach, a generic assignment node cannot be used for both C/Java (where assignments are expressions) and in Lua/Python (where they are statements). We solve many problems with the new idea of **sort injections**. Sort injections are deceptively simple: just add an AssignIsExpression node to C and an AssignIsStatement node to Python. Yet they complete sum-of-signatures by modularly specifying what *edges* may be in an AST, and, in their general form, they solve many of the limitations of sum-of-signatures. Thanks to these sort injections, Cubix can take a pre-existing syntax definition for a language, and generate a new representation of the language which is fully isomorphic to the original, but replaces portions of the AST with generic nodes.

With each language expressed as an IPS, we can write a transformation parameterized on the nodes and sort injections it deals with. It can then be run on any language that has these nodes and sort injections, but will give a compiler error when used on one that does not. These transformations can be further parameterized on language-specific operations such as symbol resolution, allowing us to build sophisticated multi-language transformations that can still handle many language-specific corner-cases.

1.3 Contributions

Overall, our paper introduces the following new ideas:

- We present the concept of **parametric syntax**, which allows the user to define a family of representations specific to different languages, but source-to-source transformations that can run on many of them. We further present techniques for **incremental parametric syntax**, which allows the user to achieve this with little work, given 3rd-party parsers and pretty-printers.
- We develop the concept of **sort injections**, which modularly specify which edges may be in an AST, and hence provide a typed and modular way to intermix language-specific and generic fragments.
- We present an algorithm for automatically converting a data type into the sum-of-products representation and its implementation in the `comptrans` code-generation library.

We use these ideas to produce the following results.

- We demonstrate using incremental parametric syntax to define a representation for C, Java, Python, JavaScript, and Lua. We show how we can define transformations that can run on all of them, including a realistic whole-program refactoring tool and complicated transformations based on control-flow, yet still achieve results that are as readable as hand transformed code.
- We present the results of a human study comparing the output of our transformations against hand-transformed code. These were identical 20% of the time, and, of the rest, judges gave ours higher average scores for readability.
- We present the RWUS (Real World, Unchanged Semantics) suite, consisting of 50 programs across 5 languages randomly drawn from top GitHub projects, together with test suites thorough enough to detect almost any modification that changes program semantics.
- We present the IPT ("Interprocedural Plumbing Transformation") tool, a whole-program refactoring for threading variables through chains of function calls, as in the Dropbox and Facebook stories. We show how we used Cubix to simultaneously build this tool for 5 languages.

1.4 Limitations

This paper is the first to present language-parametric transformations that work on multiple real languages, greatly surpassing previous attempts which were built for DSLs and language subsets such as Lämmel [2002]. Nonetheless, we have not solved all problems relating to language-parametric tools. Here are several non-goals of this paper:

First, we do not address usability. Cubix is not a tool for the lay-programmer. New undergrads on the project take about two months part-time to learn enough generic programming to begin using the system. Furthermore, we emphasize that this paper addresses only the "1-to-n" problem of extending a transformation to many languages. It is already very hard to bring a transformation to 100% correctness for one language, let alone 5; this is why there are only 4 transformations in this paper.

Second, this paper focuses on techniques for transformation, not analysis or specific transformations. The contribution of this paper is the Cubix framework and the techniques used to make language-parametric transformation possible, not the example transformations in this paper. These transformations take the results of program analyses as input, and so our work dovetails with techniques for multi-language analysis, but we do not address analysis in this paper. Note also that it is easy to integrate a language-parametric transformation with multiple language-specific analyses, so long as they provide a common interface.

Third, we have done no performance-engineering. The current implementation has substantial overhead, though we have lots of ideas for optimization, and it's still fast enough to transform and run all 2782 JavaScript tests in 5 minutes on the first author's laptop.

Building language-parametric tools is a long-term goal. There is still work to be done on verifying language-parametric transformations, using work on modular type systems to create type-aware transformations, dealing with differing memory models, preservation of formatting, etc.

2 OVERVIEW

In this section, we show how our approach allows constructing language-parametric transformations, and the work required to add support for a new language. In Section 2.1, we explain the construction of a transformation called "declaration hoisting," and how it is configured to run on several languages. Section 2.2 then explains how to create an incremental parametric syntax for C. In the language of Section 1.2, Section 2.1 defines $\mathsf{transform}$, while Section 2.2 defines $\mathsf{Remainder}_C$, $\mathsf{decompose}_C$, and $\mathsf{recompose}_C$.

2.1 An Elementary Hoisting Transformation

In this section, we describe the construction of a simplified transformation for *declaration hoisting*, and how with a small amount of configuration, we can apply it to C, Java, and JavaScript. This transformation showcases the versatility of our approach: although it totals only 22 lines for the transformation plus 30 lines for the language-specific code, it handles multiple language corner-cases, and achieves a high pass rate on the compiler validation test suites. Full code for the general portion is given in Section 4.3.

The declaration hoisting transformation moves all variable declarations to the top of the scope, using normal assignments to initialize them. The end result is similar to how C89 requires programs to be written. Figure 1 gives an example C program and its hoisted version. The elementary hoisting transformation of this section is a simplified version of the hoisting transformation in our benchmarks (5.2), which also supports Lua and handles shadowing. Neither supports Python because Python lacks variable declarations.

```
1  int f(int a,int b,int s) {
2    int t1 = 0, t2 = 1;
3    if (s) {
4      int r1 = t1*a+t2*b;
5      return r1;
6    }
7    int r2 = t2*a+t1*b;
8    return r2;
9  }
```

```
1  int f(int a,int b,int s) {
2    int t1, t2; int r2;
3    t1 = 0; t2 = 1;
4    if (s) {
5      int r1;
6      r1 = t1*a+t2*b;
7      return r1;
8    }
9    r2 = t2*a+t1*b;
10   return r2;
11 }
```

Fig. 1. An example of hoisting a C program

Setting the syntactic constraints. The user first writes a type signature declaring the general syntactic constructs a language must have to use this transformation: variable declarations, assignments, blocks, and identifiers. The type signature also requires that assignments and variable declarations must be valid members of blocks — these are *sort injections*, as described in Section 2.2. Figure 14 in Section 4 gives the code listing these constraints.

Language-specific operations. Variable initializations and assignment RHSs can be different. A Java array initialization `int[] x = {1,2,3};` becomes `x = new int[] {1,2,3};`. C variable declarators have different abstract syntax from C lvalues. To deal with these, the transformation takes as a parameter two language-specific operations, `varInitToRhs` and `varDeclBinderToLhs`.

Writing the transformation. The transformation traverses every block in the program. At each block, it checks if each block item is a variable declaration. If so, it splits the declaration into one without initialization, and into a sequence of zero or more assignments. The extracted assignments are inserted where the variable declarations lay previously, while the extracted variable declarations are prepended to the front of the block. Figure 15 in Section 4 gives this code verbatim.

Dealing with language subtleties. The hoisting transformation deals with several subtleties through the language-specific operations, but we give another one here: In JavaScript directives such as `"use strict";` must be placed at the top of a block to have effect; hoisting something above it can break the code. Perusing the spec, we saw directives are essentially treated as a separate kind of syntax, so we modified the representation of JavaScript to store them separately. This fixed bugs in multiple transformations. More examples are given in the figures in Section 5.2.

While simple, the elementary hoisting transformation in this section runs on three languages, deals with multiple language subtleties, and has a 98.4% pass rate on compiler test suites (compared to the 100% pass rate of the real version). Overall, these techniques allow transformations for different languages to share code to the extent that the two languages are syntactically similar. The rest of this paper gives more interesting transformations that also make use of static analysis and control-flow information.

2.2 Modularizing C

In Section 2.1, we outlined how to build a hoisting transformation which works on any language that contains some common notion of variable declarations, assignments, and blocks. We now show how to construct an incremental parametric syntax for C, in which parts of the language are recast in terms of these common components.

Our approach gives a language three representations. The starting point is some already-existing syntax definition of the language from a 3rd-party library, with its accompanying parser and

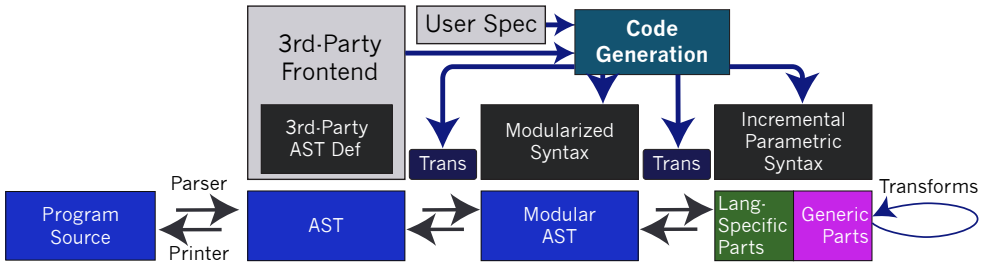

Fig. 2. Architecture of Cubix

pretty-printer. For C, we use Haskell's `language-c` library [Huber 2016], which defines C's abstract syntax as a set of mutually recursive algebraic data types like CExpression and CAssemblyStatement. Next is the "modularized" representation, which gives the exact same set of data types, but as independent signatures that do not reference each other. The sum of these signatures is isomorphic to the `language-c` abstract syntax definition. This make it easy to sum together a different set of signatures, replacing some of the C-specific data types with generic ones, yielding the third representation, the incremental parametric syntax. These three representations are mutually isomorphic, and translations between them are derived mostly automatically: the user only writes code to translate between removed node types and their generic equivalents. Figure 2 depicts how the representations and translations are generated, and how a program is transformed through each of them at runtime.

The remainder of this section explains how to construct an incremental parametric syntax for C.

Modularized representation. For each algebraic data type in the C abstract syntax, the user must generate a new data type representing nodes of that sort inside an arbitrary AST (a **signature** for that node). Combining these give a new representation identical to the original, but made of independent components. The user generates these definitions completely automatically, using the Template Haskell code-generation engine. Section 3.1 explains how we represent and combine signatures, while Section 3.4 explains the data type transformation in more detail.

Incremental parametric syntax: Nodes. The hoisting transformation is built on general components for assignments, variable declarations, and blocks. The user will need to replace these components of C, but no others, with their corresponding generic components, yielding the incremental parametric syntax. This is incremental because the user will revisit this step as more components of C need to be genericized to support new transformations.

To genericize these components, the user first compares their definitions in the C specification to the specification of the generic components, making sure the latter can model the former. To customize the generic VarDecl node to C, the user must create a new node of sort VarDeclAttrsL containing the C-specific components of a variable declaration (type and storage specifiers, assembly name, and attributes). The user does similarly for a couple other C constructs.

IPS: Sort injections. The user now finishes customizing the generic components to C by specifying where they fit in the C syntax. The user indicates that generic assignments may be used as C expressions, while C expressions form the RHS of assignments. The user does this by e.g.: generating a AssignIsCExpr node. This establishes an injection from terms of sort Assign to terms of sort CExpr, which we call a **sort injection**. The user does similar to place the other generic nodes within the C syntax. Cubix generates nodes witnessing these sort injections

IPS: Putting it together. The user now defines the incremental parametric syntax for C by writing a couple lines of Template Haskell that takes the list of signatures in the modularized syntax, subtracts the replaced nodes, and adds the generic components and sort injection nodes. This code

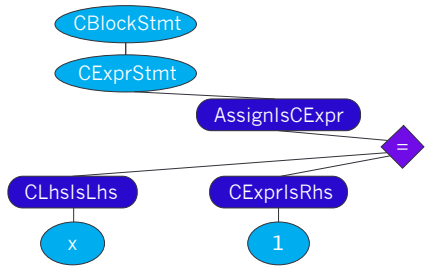

Fig. 3. A term in the incremental parametric syntax for C. The ellipses (light blue) represent language-specific nodes; rhombi (purple) represent generic nodes; rounded rectangles (dark blue) represent sort injection nodes.

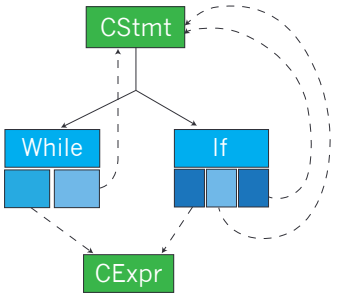

Fig. 4. Fragment of a typical representation of C. The solid arrows represent the instance-of relationship; dotted represent containment.

is given in Figure 13 in Section 4.1. The sum of these signatures is the signature for the IPS for C, and the terms of this signature are given by its type-level fixpoint. These terms resemble Figure 3, showing the mixture of C-specific and generic nodes, with sort injections between them.

IPS: Translations. The user writes instances of the trans and untrans operators between the nodes that have been removed from C, and the generic ones that replaced them. Generic programming deals with the nodes shared between the IPS and the modularized syntax, giving translation functions between the two representations. Our actual implementation of these translations for C totals 130 lines of Haskell code, about 40 of which are boilerplate.

3 CORE IDEAS

In this section, we explain the core new ideas that make our language-parametric transformations possible. Section 3.1 gives background on modular syntax, used in the rest of this section Section 3.2 presents the terminology and goals of incremental parametric syntax. We achieve this through the concepts in the following sections: Section 3.3 presents sort injections, and Section 3.4 explains the translation of a syntax into its modularized version.

3.1 Background: Data Types à la Carte

The basic idea of modular syntax is simple: languages should be defined by a set of nodes, and the same node can appear in many languages. So, a transformation to swap the two branches of an if-statement should be runnable on any language with if-statements.

Unfortunately, in common representations of syntax, whether as an algebraic data type (ADT) like the fragment in Figure 4, or as a set of classes, this is not possible. The problem is mutual recursion between types. A C if-statement contains C expressions, which can contain C statements. So, the node for C if-statements is tied to definitions for all other C statements. The structure of

```
1  data Add e = Add e e
2  data Val e = Val Int
3  data (f :+: g) e = Inl (f e) | Inr (g e)
4  data Term f = Term (f (Term f))
5
6  type ExpSig = Add :+: Const
7  type Exp = Term ExpSig
8
9  addExample :: Exp
10 addExample = Term (Inl (Add (Term (Inr (Val 118))) (Term (Inr (Val 1219)))))
```

Fig. 5. Using data types à la carte to present the expression 118+1219, with addition and constant nodes defined in separate fragments.

code follows the structure of data, and so a traversal written over this type will also be coupled to all C statements.

Even without the mutual recursion, trouble arises as soon as one uses a fixed type like $C \to C$ or Java $\to$ Java for a program transformation. The reason goes back to the basic theory of subtyping. Producer functions of type $A \to C$ are *covariant* in C, meaning new cases can be added to C without changing the function. Consumer functions of type $C \to A$ are *contravariant* in C, meaning cases can be removed from C without changing the function. But functions of type $C \to C$ are *invariant*, meaning the code will break if any cases are added or removed from the language. Techniques such as the visitor pattern can help, but introduce new limitations (discussed by Lämmel et al. [2003]), and do not allow for a multi-language transformation so long as the types are tied together. Switching to a dynamically-typed language also does not help; removing the types does not remove constraints over the data.

What does help is removing the recursion from the syntax definitions, and using parametric polymorphism for the types. Mathematically, an ADT is defined in three stages: first data is tupled into a constructor; then many constructors are summed into a signature, a list of node types with unspecified children; and then a fixpoint is taken over the signature, yielding recursive trees. The idea of the sum-of-signatures representation, known in the functional programming community as data types à la carte (DLC) [Swierstra 2008], is to defer the fixpoint operation. The programmer instead programs against signatures, which are not recursive, and can be modularly combined.

In DLC, a signature takes the form of a data type similar to conventional abstract syntax, but where all recursive terms have been replaced with a type variable, so that the type of children may be specified later. Each signature may represent an independent fragment of a language; these signatures may be freely summed into a signature for an entire language, and then closed recursively, as depicted in Figure 6. Figure 5 shows an example of a term written in DLC, taken from Swierstra [2008].

Data types à la carte generalizes easily to multiple sorts: have a type variable for terms of sort Stmt, another for terms of sort Exp, etc. This unfortunately makes it difficult to add new sorts, or to have languages with different numbers of sorts. The insight of Yakushev et al. [2009] is to merge these into a single higher-order type variable t. Subscripting t with various labels gives the type of terms of a certain sort: t_{Stmt} represents terms of sort Stmt, t_{Exp} represents terms of sort Exp, etc. But t itself is a single variable, representing terms of all sorts. Figure 10 shows signatures following this pattern.

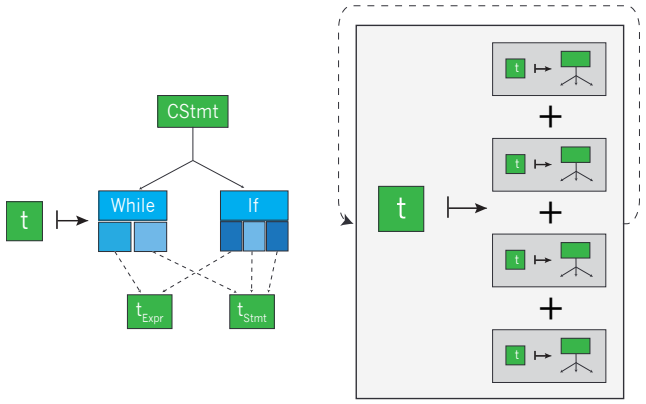

Fig. 6. In DLC, a language is represented by a list of subsignatures like the one on the left. Each signature has a type variable for subterms, in lieu of self-reference. The subsignatures are combined into a signature for the whole language, which is then closed by specifying that allowed subterms of terms of this signature are other terms of this signature (right).

3.2 Incremental Parametric Syntax

As explained above, functions of type $C \to C$ have a type which is *invariant* in C. That is, in general, the code for any function that consumes and produces a value of type C will break when the definition of C is modified. So, instead of using a fixed type, the way to write a function that can transform many data types is with parametric polymorphism. For instance, the sort function of type $\forall x.\text{Ord x} \Rightarrow [x] \to [x]$ works over lists of any data type that supports comparison, and, after inlining, is just as efficient as a sort function written for each data type. Our goal is to bring this combination of generality and specialization to program transformation.

Let $F_1, \ldots, F_n$ be fragments that may be contained in many languages. We define a **parametric syntax** $\mathcal{S}$ for a language as any representation that supports an operation $\prec$ such that a transformation over any language containing $\overline{F_i}$ may be written $\forall x.\overline{F_i \prec x} \Rightarrow x \to x$. This gives a name to previous work: any language written in DLC is a parametric syntax.

But the drawback of previous incarnations DLC and other forms of modular syntax is that language definitions in those styles are what we term a **fully parametric syntax**, meaning that the syntax must be written entirely in terms of generic fragments.

More formally, a fully parametric syntax is any representation satisfying:

(1) There is some combination operator $\bowtie$ which merges fragments. The $\bowtie$ operation must satisfy the property: if $F \prec G$ or $F \prec H$, then $F \prec (G \bowtie H)$.
(2) Each syntax definition is built entirely by combining generic fragments. That is, $\mathcal{S}$ is a fully-parametric syntax if it can be written $\mathcal{S} \doteq G_1 \bowtie \ldots \bowtie G_m$, where each $G_i \in \{F_1, \ldots, F_n\}$.

Defining a fully parametric syntax for a language requires a large amount of up-front labor. Incremental parametric syntax lowers this initial barrier.

We say that $\mathcal{S}$ is an **incremental parametric syntax** if there is a non-parametric syntax $\mathcal{T}$ and a "fragment removal" operator $\setminus$ such that $\mathcal{S}$ may be expressed:

$$\mathcal{S} \doteq (\mathcal{T} \setminus F_1 \setminus \cdots \setminus F_m) \bowtie G_1 \bowtie \ldots \bowtie G_n$$

An incremental parametric syntax allows the user to start with a pre-existing syntax definition, replace some components with their generic equivalents, and then write transformations against the generic components. Given the complexity of a production language, this approach is necessary

for getting a language-parametric transformation running on real languages in a reasonable amount of time.

In our instantiation of incremental parametric syntax, we use the signature subsumption and sums from data types à la carte to provide the fragment subsumption ($\prec$) and $\bowtie$ operators. We use new ideas for the $\setminus$ operator: convert the existing syntax into a sum of language-specific signatures (Section 3.4), and then use signature subtraction. Additionally, to add generic fragments, one must also add new nodes to reshape the grammar to accept them (Section 3.3),

Parametric syntax closely relates to the Expression Problem [Wadler 1998], which concerns being able to separately extend a language with new terms and new operations. Any incremental parametric syntax is also a solution to the Expression Problem, as it allows a language to be extended with new terms and operations. However, a solution to the Expression Problem need not allow for expressing multiple languages. As parametric syntax is our name for a family of existing approaches, discussion of how parametric syntax solves the Expression Problem can be found in the DLC paper [Swierstra 2008].

3.3 Sort Injections

Using data types à la carte, we can modularly specify which nodes may be in a language, and replace them with generic ones. However, similar nodes in different languages may interact differently with the rest of the language. Assignments are expressions in C/Java and statements in Lua/Python. Most languages have various assignment operators like $+=$, but Lua does not. Rarely will a generic node be an exact fit for a construct already in a language. Instead, it must be customized for that language.

We solve this with sort injections. A **sort injection** from A to B is an injective function from terms of sort A to terms of sort B, together with its partial inverse. C and Java have a sort injection from Assign to CExpr and JavaExpr respectively, while Lua and Python have ones from Assign to their respective statement sorts. And all languages except Lua have a sort-injection from their respective language-specific assignment-operation sorts to the generic assignment-operation sort.

The most straightforward way to provide such a sort injection is via a **sort injection node**, an unary production of sort B with a single child of sort A. Figure 7 gives an example sort injection and node from generic assignments to C expressions.

So, while DLC modularizes which nodes may be in a languages, sort-injections modularize the edges. Adding the AssignIsCExpr node from Figure 7 to a syntax definition is equivalent to allowing a parent-child edge from anything that contains C expressions to assignments.

Sort injections also serve an additional purpose: abstracting over intermediate nodes. In all supported languages, assignments may be used as top-level items in blocks. However, this occurs through a chain of language-specific nodes. Block statements in C may not be assignments directly, but they can be ordinary statements, which may be expression statements, which may be assignments. And, in the 3rd-party JavaScript frontend used by Cubix, there are actually two (semantically-equivalent) ways that assignments may be statements. All this can be abstracted into the constraint: there is a sort injection from assignments to block items. Figure 8 illustrates this example.

In a transformation such as Hoist that works on languages with a sort injection from assignments to block items, the transformation has the ability to place assignments into blocks, and to check if a block item is an assignment. So one can think of this transformation as working not on the original tree but on a "blown-down" tree, which only contains these generic nodes. Figure 9 shows an example of a blown-down tree. This is similar to the theory views seen in Maude [Clavel et al. 2002], and to the homeomorphic embedding in term rewriting [Baader and Nipkow 1999].

```
1  data AssignIsCExpr t l where
2    AssignIsCExpr :: t AssignL → AssignIsCExpr t CExprL
3  instance (AssignIsCExpr ∈ f) ⇒ InjF (Term f) AssignL CExprL where
4    injF = AssignIsCExpr
5    projF x = case project x of
6      Just (AssignIsCExpr x) → Just x
7      _                      → Nothing
```

Fig. 7. Sort injection node and its associated sort injection

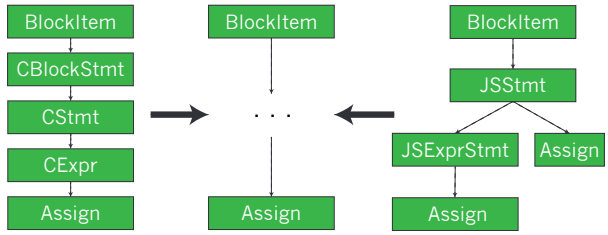

Fig. 8. Sort injections from Assign to BlockItem

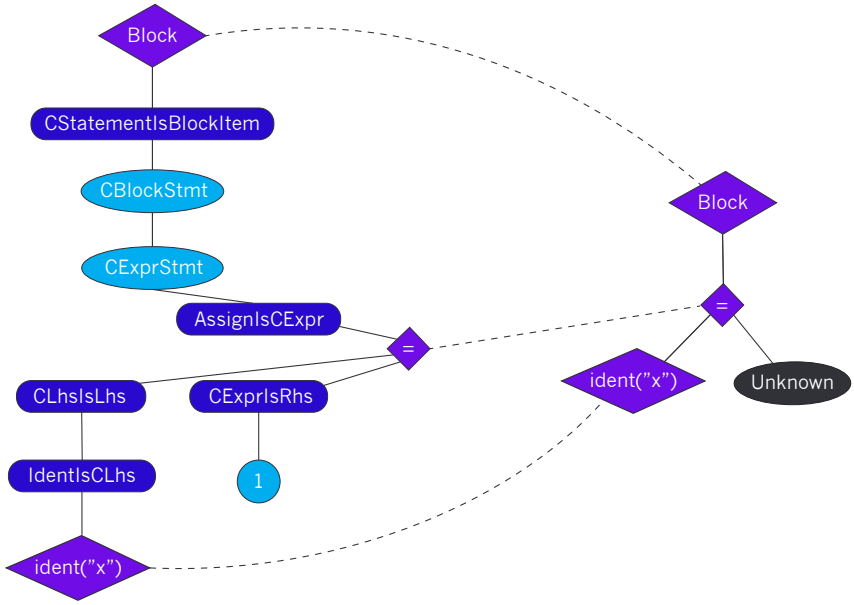

Fig. 9. Blowing down a tree

3.4 Modularizing a Syntax Definition

The preceding sections gave some of the techniques of incremental parametric syntax; we now show how to convert an existing syntax definition so that it can be incrementally generalized. The key idea is to transform an existing syntax definition $\mathcal{T}$ into the combination $F_1 \bowtie \ldots \bowtie F_m$. This provides the final component of incremental parametric syntax, as $\mathcal{T} \setminus F_i$ can be defined by simply removing F_i from the combination.

The ADT modularization transformation is most easily explained by an example: it transforms the ADTs on the left side of Figure 10 into the generalized algebraic data types (GADTs) on the

```
data Arith = Add Atom Atom

data Atom = Var String
          | Const Lit
          | Parens Arith

data Lit = Lit Int
```

```
data ArithL; data AtomL; data LitL
data Arith t l where
  Add :: t AtomL → t AtomL
                 → Arith t ArithL
data Atom t l where
  Var    :: String    → Atom t AtomL
  Const  :: t LitL    → Atom t AtomL
  Parens :: t ArithL  → Atom t AtomL
data Lit (t :: * → *) l where
  Lit :: Int → Lit t LitL
```

Fig. 10. Example input (left) and output (right) of comptrans.

```
data (:+:) f g t l = Inl (f t l) | Inr (g t l)
data Term f l = Term (f (Term f)) l
type LangSig = Arith :+: Atom :+: Lit
type LangTerm = Term LangSig
```

Fig. 11. Combining the fragments of Figure 10

right. The GADTs stand independently, with no recursion between them. Instead of a recursive reference to the Arith type, for example, the type t ArithL can be read "Terms of sort ArithL, which will be specified later." But when those terms are specified, and the independent types are combined back together, the result type Term (Arith :+: Atom :+: Lit) ArithL is isomorphic to Arith. Figure 11 shows how these types are combined.

In our instantiation of incremental parametric syntax, this means converting a syntax definition into DLC. For syntax definitions given as mutually recursive algebraic data types, this is quite easy to do. The recursive knot is already tied in a separate step in the metatheory; the ADT modularization transformation just puts that in code as well.

The transformation generalizes easily from this example. We give a full formal definition in Appendix Aof the arXiv version of this paper [Koppel et al. 2018], and implement it in our `comptrans` tool.

The modularized representation has several other benefits, even when writing transformations for only one language. For instance, it allows giving a type Term Sig l → Term Sig l to sort-preserving rewrites that can be applied to terms of any sort, and also enables many generic-programming techniques. See Bahr and Hvitved [2011] for a full discussion.

4 IMPLEMENTATION

We have implemented our approach in the Cubix system, named for a fictional robot composed of modular parts that can be reassembled for many purposes. Cubix is organized as a collection of libraries which assist in building incremental parametric syntaxes and language-parametric transformations. Our implementation totals approximately 13,000 lines of Haskell, providing support for five languages and several transformations. We build heavily on the `compdata` library of Bahr and Hvitved [2011] for modular syntax, and extend it with support for sort injections and the `comptrans` library for converting a third-party syntax definition into modular syntax. We provide generic language components, and modules for labeled terms, control-flow graphs, and higher-order tree traversals.

```
data MultiVarDeclAttrsL; data VarInitL; data MultiVarDeclL;
data OptVarInitL; data VarDeclAttrsL; data VarDeclL;
data AssignOpL; data AssignL; data LhsL; data RhsL
data OptVarInit t | where
  JustVarInit :: t VarInitL → OptVarInit t OptLocalVarInitL
  NoVarInit   :: OptVarInit t OptVarInitL
data VarDecl t | where
  VarDecl :: t VarDeclAttrsL → t VarDeclBinderL
          → t OptVarInitL → VarDecl t VarDeclL
data MultiVarDecl t | where
  MultiVarDecl :: t MultiVarDeclAttrsL → t [VarDeclL]
               → MultiVarDecl t MultiVarDeclL
data Assign t | where
  Assign :: t LhsL → t AssignOpL → t RhsL → Assign t AssignL
```

Fig. 12. Generic nodes to model vardecls and assignments

```
do let cSortInjections = [''CExprIsRhs, ''AssignIsCExpr, ...]
   let names = (cSigNames \\ [mkName "Ident", ...])
         ++ cSortInjections ++ [''VarDecl, ''P.Ident, ''Assign, ...]
   runCompTrans (makeSumType "MCSig" names)
```

Fig. 13. Generating the incremental parametric syntax

The code is split between approximately 5000 lines in our language implementations, 2300 in our transformations, 1200 in `comptrans`, 3900 in our other libraries, and the rest in our driver, miscellaneous code, and minor extensions to 3rd-party libraries. Note that our language implementations do contain a lot of code clones, due to the limits of metaprogramming in Haskell.

The rest of this section discusses Cubix in more detail. Section 4.1 describes implementing an IPS in Cubix. Section 4.2 describes how to implement transformations, and Section 4.3 gives example code. Section 4.4 discusses how to generalize Cubix beyond Haskell and the 5 target languages.

4.1 Languages

As shown in Figure 2, to add support for a language, the users selects a 3rd-party frontend, and then constructs two derived representations.

Creating the modularized syntax. There are three steps to creating the modularized syntax. Because the modularized syntax is identical to the original, but in a different form, this is completely automatic.

First, for each algebraic data type in the original AST, the user must create a language fragment signature similar to the one in Figure 10. `comptrans` generates this code automatically using Haskell's compile-time code-generation engine, Template Haskell [Sheard and Jones 2002]. For instance, the command to do this for C is `runCompTrans (deriveMultiComp ''CTranslationUnit)`, as `CTranslationUnit` is the root of the C type.

Second, the user sums these language fragments into a signature for the language. For C, the command is `runCompTrans (makeSumType "CSig" cSigNames)`. The user may now manually declare types in terms of `CSig`, such as the type of C terms **type** `CTerm = Term CSig`, and the signature of labeled C terms **type** `CSigLab = CSig :&: Label`.

Finally, another command is used to generate the translations between the `language-c` representation and the modularized representation.

```
type HasSyntax f = (VarDecl ≺ f, MultiVarDecl ≺ f
  , OptVarInit ≺ f, Ident ≺ f, Assign ≺ f, AssignOpEquals ≺ f
  , Block ≺ f, ListF ≺ f, ExtractF [] (Term f))
type CanHoist f = (HasSyntax f, VarInitToRhs (Term f)
  , VarDeclBinderToLhs (Term f), HTraversable f
  , InjF f MultiVarDeclL BlockItemL, InjF f AssignL BlockItemL)
```

Fig. 14. Constraints for the elementary hoist transform

Designing a Library of Generic Components. Designing a generic language component takes serious thought: it must be possible to instantiate it in a way that models the corresponding construct in every language under consideration.

These come in the form of (completely-standalone) manually-written signatures. Figure 12 shows Cubix's definitions for generic variable declarations and assignments, which we designed to model the corresponding constructs in C, Java, JavaScript, Lua, and Python. The empty data declarations like **data** LhsL denote sorts, while the others are generic nodes. This component comes with many knobs. By providing C-specific nodes of sort VarDeclAttrsL and MultiVarDeclAttrsL, it can model declarations like `const int x = 1, *y;`. By providing empty nodes of those sorts, it can model Lua and Python variable declarations.

Creating the IPS. The user must now decide how to instantiate the generic components in Figure 12 to model their language-specific counterparts. For instance, in C, assignments are expressions, and expressions are assignment right-hand sides. The user specifies this by generating a sort injection from AssignL to CExprL and from CExprL to RhsL, and does similar for LhsL and AssignOpL.

The user is now ready to define the IPS for the language. This is done by expressing the old signature as a compile-time list of symbols, and literallly removing the language-specific components and adding the generic ones. Figure 13 gives the code used to generate the IPS C signature, MCSig

Finally, the user must write a translation from the modularized syntax to the IPS. They need only write code for the cases where the syntaxes differ, i.e.: to replace language-specific nodes with generic ones.

This completes the process depicted in Figure 2.

Other support. Some transformations may require other language-infrastructure, such as a control-flow graph generator. The IPS representation makes it easy to share code across languages; our 5 CFG-generators average 101 LOC.

In our experience, creating an incremental parametric syntax for a new language takes 1-2 days. We have implemented support for C, Java, JavaScript, Lua, and Python, using the parsers, pretty-printers, and syntax definitions from the Haskell libraries `language-c` [Huber 2016], `language-java` [Broberg 2015], `language-javascript` [Zimmerman 2016], `language-lua` [Ömer Sinan Ağacan and Mertens 2016], and lastly `language-python` [Pope 2016]. Because of problems with the parser for `language-java`, we instead use a Java parser written in Java, the javaparser.org parser [van Bruggen 2016], and translate its results into the `language-java` AST. Despite their names, these libraries were all implemented independently by different authors, and share no common infrastructure beyond standard libraries. We fixed bugs in all of their pretty printers but were otherwise not involved with their development. Some of our fixes have yet to be merged upstream.

4.2 Transformation Support

A language-parametric transformation makes limited assumptions about its target language. This is done by parameterizing the transformation over operations on the nodes and terms of the language, given in the form of Haskell typeclasses.

The constraints for the elementary hoisting transformation, CanHoist in Figure 14, depicts the full spectrum of such operations. The elementary hoisting transformation can run on any language that satisfies these constraints, and gives a compile error on any that do not. First, there are constraints that the language contain generic nodes. This is given as the $\prec$ constraint from `compdata`, which provides an injective function from the generic node to terms of the language, inject, and its partial inverse, project. As a second class, the InjF constraints are sort injections as discussed in Section 3.3. Finally, VarDeclBinderToLhs and VarInitToRhs provide the language-specific operations of elementary hoisting, discussed in Section 2.1. Overall, these constraints allow a transformation to make a limited set of assumptions about its target languages, allowing it to handle the intricate details of many languages while maintaining a high level of generality.

There are also a couple more technical constraints. The interface HTraversable from compdata interface offers generic tree-traversal operations. MaybeF and ListF provide tree nodes representing optional nodes and lists of nodes, so a node representing a list of block items may have sort [BlockItemL]. There are then operations extractF and insertF to convert between values of type Term f [I] (term of sort "list of I") and values of type [Term f I] (list of terms of sort I).

We have built a library of *strategy combinators* [Lämmel and Visser 2002] called `compstrat`. With strategy combinators, the user can turn a set of single-node rewrites into a complicated traversal pattern in a single line of code. `compstrat` provides similar functionality to other strategy combinator libraries such as Scrap Your Boilerplate [Lämmel and Jones 2003], Strafunski [Lämmel and Visser 2003], and KURE [Gill 2009].

We have also built miscellaneous other infrastructure to support our transformations. The most interesting of these is the control-flow based inserter. Inserting a statement before a loop condition causes it to be placed before the loop, before the end of the loop, and before every continue statement.

4.3 Example: Implementing the Elementary Hoisting Transformation

This section presents the full implementation of the elementary hoisting transformation from Section 2.1. Figure 15 gives the code; Figure 14 showed the CanHoist constraint. We omit the 30 lines of code giving the three language-specific instances of VarInitToRhs and VarDeclBinderToLhs.

The code implements the algorithm described in Section 2.1. Execution begins at elementaryHoist, which runs hoistBlockItems over every block. It does so by using `compdata`'s transform function to run hoistInner over every node, which uses project to test if a node is a block. Later, the sort injections are used via projF and injF to operate on the subset of BlockItem's that the transformation knows about.

In this example, we have tried to avoid many of the vagaries of Haskell syntax as well as more advanced features of Cubix. Nonetheless, some advanced features are present. The sum-of-signatures approach distinguishes between nodes, which may lie in an arbitrary AST, and terms, which are tied to a single language. The vanilla data constructors of Figure 12 like Assign construct nodes of a signature fragment, while their primed variants like Assign′ construct and pattern match on terms. We explained the extractF and insertF functions in Section 4.2; these are used to implement the liftF and mapF functions, which are used to operate on trees of type Term f [I]. Finally, the syntax `f (view → Just x) = ...` is a Haskell *view pattern* [Wadler 1987] which is syntactic sugar for `f t = case view t of Just x → ...`, with pattern match failure proceeding to the next case.

```
declToAssign :: (CanHoist f) ⇒ Term f MultiVarDeclAttrsL → Term f VarDeclL → [Term f BlockItemL]
declToAssign mattrs (VarDecl' lattrs b optInit) = case optInit of
  NoVarInit'          → []
  JustVarInit' init → [injF (Assign' (varDeclBinderToLhs b) AssignOpEquals' (varInitToRhs mattrs b lattrs init))]

removeInit :: (CanHoist f) ⇒ Term f VarDeclL → Term f VarDeclL
removeInit (VarDecl' a n _) = VarDecl' a n NoVarInit'

splitDecl :: (CanHoist f) ⇒ Term f BlockItemL → ([Term f BlockItemL], [Term f BlockItemL])
splitDecl (projF → (Just (MultiVarDecl' attrs decls)))
          = ([injF (MultiVarDecl' attrs (mapF removeInit decls))], concat (map (declToAssign attrs) (extractF decls)))
splitDecl t = ([], [t])

hoistBlockItems :: (CanHoist f) ⇒ [Term f BlockItemL] → [Term f BlockItemL]
hoistBlockItems bs = concat decls ++ concat stmts
  where (decls, stmts) = unzip (map splitDecl bs)

elementaryHoist :: (CanHoist f) ⇒ Term f l → Term f l
elementaryHoist t = transform hoistInner t
  where hoistInner :: (CanHoist f) ⇒ Term f l → Term f l
        hoistInner (project → (Just (Block bs e))) = Block' (liftF hoistBlockItems bs) e
        hoistInner t                               = t
```

Fig. 15. Implementation of the elementary hoist transformation

Table 1. Various term types in Cubix

Type signature	Description
`Term f AssignL`	Assignments in any language
`Term MJavaSig l`	Java terms of any sort
`(Assign < f) ⇒ Term f IdentL`	An identifier in any language that contains generic assignments
`Term f (StatSort f)`	A statement in any language. The statement sort is language-specific.
`(InjF f IdentL PositionalArgExpL, CallAnalysis f) ⇒ Term f IdentL`	An identifier in any language which supports a call analysis, and where identifiers may be used as ordinary arguments to functions

4.4 Choices of Target and Implementation Languages

When we speak about Cubix, we always find people who want to use it or something like it for their applications. What would it take to implement a system like Cubix in a different language? And what about supporting other languages, such as ML or Prolog or Haskell itself?

What do we gain from these fancy types? Cubix's implementation of incremental parametric syntax uses some rather advanced type system features. Case in point: the current implementation uses a total of 32 GHC extensions. What's the benefit of all this, and can it be replicated in a language other than Haskell?

There are two primary benefits. The first is the precise typing. The second is dispatch: the compiler uses the type information to choose appropriate language-specific and sort-specific operations. Both are indispensible for building multi-language tools. And their synthesis allows generic programming.

Consider the example types in Table 1. They show how, using the `Term f l` type, developers can restrict operations to certain sorts of terms, languages, and properties of the language. These restrictions are all enforced by the compiler.

Without these types, it's still quite easy to write a function that can accept many kinds of terms, by giving them all a single "Node" type. This is the dynamically-typed or "generic node" approach, used in many language workbenches. This is not enough to get language-parametric transformation, as removing the types does not remove the network of constraints between AST nodes. There must be the extra step of converting part of the tree to some common form, as done in IPS. And these conversions introduce massive room for errors.

Our own experience attempting this kind of generic programing in JavaScript, as well as fixing type errors during normal Cubix development, makes us pessimistic about trying it without precise types. It's far too easy to e.g.: attempt to use an assignment as an expression, when that is not legal in every language.

The second major benefit is dispatch. Consider writing a transformation that works on any language where functions can be turned into lambdas. If we were to implement this transformation in a language without typeclasses, we would make the transformation take a "turn function into lambda" operation as a parameter. This operation would then need to be transitively supplied to every piece of the transformation that needed it.

Conversely, in Haskell, we'd simply add a condition to the constraints for the transformation, as in Figure 14, and it would be propagated to all components. And when attempting to call this transformation on a specific language, the compiler will automatically find and supply the correct instance of the "turn function into lambda" operation.

In other words, it is very easy to write a generic operation that includes language-specific pieces. Doing this at a smaller level is generic programming. For instance, `subterms e :: [Term f IdentL]` gives all identifiers contained in `e`. The equivalent code in a language without typeclasses would be something like `subterms(e, Filters.checkSort(SORT_IDENT))`. Meanwhile, Haskell automatically supplies `subterms` with the correct sort-classification check by looking at the types. So, this representation is useful even when only working with one language.

IPS in other languages. Our implementation of incremental parametric syntax relies heavily on three distinctive features. We discussed the use of type classes above. The `Term f I` type relies on GADTs to work (else `Term f IdentL` could not be fundamentally different from `Term f AssignL`). And we've used Haskell's built-in code generation, Template Haskell, throughout this paper. Haskell is the only language we know of that supports all three features. But even Haskell is not a perfect language for building this kind of system, and we still have a wishlist of language features that would make Cubix development much easier (e.g.: pattern matching that works better with modular syntax).

We can envision a Cubix-like framework in a language without any of these three features. It would use an extra-linguistic tool to generate boilerplate code. Users would pass in operations manually in lieu of typeclasses, at some inconvenience. But without GADTs, we see only two options, neither of them appealing: either write a custom type-checker/analyzer, or face the pitfalls of dynamically-typed terms.

Supporting more languages. To share code between languages, these languages must have nodes which are similar enough to design a generic node whose semantics model all of them. Expressions are similar in C and ML, and we see no barrier writing transformations that can operate on them both. On the other hand, the execution semantics of Prolog nodes differs substantially from imperative and functional languages, and so we do not expect to be able to write C/Prolog transformations.

Integrating Cubix with a third-party parser only requires that the parser output to a Haskell ADT. We hence picked languages that already had good Haskell libraries, but it can integrate with parsers written in other languages by writing a wrapper, as we have done for Java.

```
1  void f1(char* buf) {
2    strcpy(buf, "Hello");
3  }
4  void f2(char* buf) {
5    f1(buf);
6  }
7  void f3(int len) {
8    char *buf = malloc(len);
9    f2(buf);
10 }
```

```
1  void f1(char* buf, int len) {
2    strcpy(buf, "Hello");
3  }
4  void f2(char* buf, int len) {
5    f1(buf, len);
6  }
7  void f3(int len) {
8    char *buf = malloc(len);
9    f2(buf, len);
10 }
```

Fig. 16. Input/output example of the IPT tool on C

5 EVALUATION

In the previous section, we argued that the insights of Cubix make language-parametric transformations easy to write. In this Section, we demonstrate a realistic language-parametric transformation and its application to real software, and further evaluate the following two claims:

- *Readability*: These transformations produce readable output, similar to what a human would write. They do not needlessly destroy the program's structure, as do IR-based transformations.
- *Correctness*: Despite the low effort needed per language, transformations can maintain correctness even when faced with the intricacies of multiple languages.

Additionally, because we built the tool of Section 5.1 after the rest of the work in this paper, our experience building also supports our claim that it is easy to extend an IPS as more features are needed to support new transformations.

5.1 A Realistic Whole-Program Refactoring

In this section, we present the IPT tool (**i**nterprocedural **p**lumbing **t**ransformation) for threading variables through chains of function calls, inspired by the Dropbox and Facebook stories in Section 1. We built the IPT tool as a language-parametric transformation which we developed simultaneously for all 5 languages supported by Cubix.

The IPT tool takes a method and a parameter name, and recursively has all callers pass down said parameter, asking for user approval for each change. Figure 16 presents a scenario where the end goal is to replace the call to `strcpy` within `f1` with `strncpy`, and the barrier is that the programmer is missing the `len` parameter needed by `strncpy`. He invokes the IPT tool to add an `int len` parameter to `f1`, pressing "Yes" 4 times to change lines 1, 5, 4, and 9, so that the existing `len` parameter in `f3` is passed through 2 layers of function calls to `f1`. After the IPT tool is done, the programmer can now manually change line 2 to `strncpy(buf, "Hello", len);`. The IPT tool has automated plumbing data through the system; all that is left for the programmer is to choose where it comes from, and how it's used.

Building this tool also shows that it is easy to extend an IPS to support new transformations. We had not needed a generic notion of functions for our previous transformations we implemented (see Section 5.2), so we made an incremental change to our parametric syntax. In 3 hours, we designed a generic fragment for function definitions and calls that could be instantiated to model the features of all languages under consideration. It took us 21 hours to design and implement the changes to all 5 language representations, proportional to the complexity of each language (e.g.: 5 hours to understand C declarations and their many variations). We thus obtained the incrementality benefits of IPS: we didn't need to build up-front support for functions, but could still build transformations that needed them, and we now have support for functions for all future transformations.

With the generic syntax for functions in place, building the tool itself took only 19 hours. Altogether, the extensions to Cubix averaged 5 hours per language, while the tool itself averaged another 4 hours per language.

The implementation is fairly straightforward: it maintains two queues of function calls and definitions to be modified, and prompts the user about each potential change. After modifying each function definition, it uses a static analysis to find all callers and add them to the queue. This static analysis is a parameter of each language, but the analyses for each language may use a shared implementation using techniques of multi-language analysis. Making this analysis more precise means the user will be prompted for fewer erroneous changes.

Our IPT tool is still a prototype, with a minimal command-line UI, an imprecise call analysis, and incomplete support for C function prototypes. Nonetheless, we have used it in three real case studies in Java and Python, in addition to toy programs in the other three languages.

We first used it on SimpleDB [Madden 2017], a teaching database used at several universities. SimpleDB totals 23,000 lines of Java, and 11,500 lines of tests. It frequently accesses a global `BufferPool` object by calling `Database.getBufferPool()`. We used a two-step process to eliminate this global. First, we used the IPT tool to thread a `bufferPool` parameter throughout the program. This changed all `Database.getBufferPool()` calls to instead read `Database.getBufferPool(bufferPool)`. Second, we applied a find-and-replace to the entire program to simplify them to `bufferPool`. We then manually changed entry points to the program to supply this `bufferPool` parameter. Altogether, the IPT tool modified 484 lines across 41 files, while we manually modified 50 lines across 24 files. All tests pass.

We then did two smaller case studies in Python. Flask [Grinberg 2014] is a Python web micro-framework which totals 6500 lines of Python and 5700 lines of tests. We used the IPT tool to modify the `_get_exc_class_and_code` function to take a default exception code, and propagated this parameter up several layers. The IPT tool modified 21 lines across 2 files. We only manually changed 2 lines: to use this parameter, and to supply it at the top of the chain. Tornado[Dory et al. 2012] is a Python web server owned by Facebook. It comprises 22,000 lines of Python and 16,000 lines of tests. We changed the `is_valid_ip` function to take an `accept_ipv6` parameter, and propagated this parameter up several layers. The IPT tool changed 22 lines across 9 files. We used a find-and-replace to provide a default value to many new parameters, and then changed 3 lines manually. All tests pass for both Flask and Tornado.

For all five languages, we also tested the IPT tool on a toy program consisting of three functions across three files that call each other; the tool successfully propagated a parameter through all three functions.

5.2 Benchmark Transformations

To more rigorously evaluate our system, we have implemented three smaller source-to-source transformations. These were chosen to explore the space of operations used by program transformations and to require a minimum of user input. Table 2 lists them and their line counts.

- The hoisting transformation **Hoist**, which lifts variable declarations to the top of their scope. This is similar to elementary hoisting in Sections 2.1 and 4.3, except that it also supports Lua, and uses additional machinery to avoid hoisting shadowed variables, and to deal with language special-cases such as C's structure initializers. Figure 1 gave a mundane example; Figure 17 gives an example handling a JavaScript special case. This transformation supports all languages except Python, which lacks variable declarations.
- The test-coverage instrumentation transformation **Testcov**, which prefixes each basic block in the source code with an assignment which marks that that block has executed. This produces data that could be fed into a test coverage tool. It was inspired a Semantic Designs tool which

```
1 function f() {
2   "use strict";
3   if (x) {
4       var y =1;
5   }
6 }
```

```
1 function f() {
2   "use strict";
3   var y;
4   if (x) {
5     y = 1;
6   }
7 }
```

Fig. 17. Hoisting JavaScript, showing interactions with JS's `"use strict";` pragmas and lack of inner scopes. No JS-specific hoisting code is needed, only a precise representation of JS blocks.

```
1 public static void foo(int x) {
2   if (x > 0) {
3     while(true)
4       x++;
5     // Unreachable code
6   }
7 }
```

```
1 public static void foo(int x) {
2   TestCoverage.coverage[0] = true;
3   if (x > 0) {
4     TestCoverage.coverage[1] = true;
5     while (true)
6       x++;
7   }
8   TestCoverage.coverage[2] = true;
9 }
```

Fig. 18. Test coverage for Java. A naive transformation would insert a test coverage statement on line 5 after the while loop, causing an "unreachable code" compile error. This case is supported purely through the CFG-generator, requiring no Java-specific code in the transformation itself.

Table 2. Transformations implemented and their size. Line counts are split into the core code of the transformation, plus the per-language code to support language-specific operations and customization. Line counts exclude the file prologue, i.e.: they count from the first line of code which is not an `import` statement.

Transformation	Languages Supported	Core LOC	Extra LOC per language
Hoist	C, Java, JavaScript, Lua	154	65
Testcov	All	77	25
TAC	JavaScript, Lua, Python	360	116

implements this transform separately for a dozen languages [Semantic Designs, Inc 2005]. This transformation supports all languages. Figure 18 shows an example special case for Java.

- The three-address code transformation **TAC** hoists all nested computations into temporary variables, e.g.: changing `1+1+1` into `t=1+1; t+1`. This is a deceptively complicated transformation, difficult to write at the source level for even one language. Figures 19 and 20 show a few of the complexities it supports, all handled cleanly by Cubix's general infrastructure for operator strictness and CFG-based insertion. This transformation supports JS, Lua, and Python. It does not support Java or C because declaring the temporary variables would require type inference, which in turn requires symbol-table construction, a heavyweight piece of language infrastructure.

All transformations use a mixture of generic and language-specific code. However, the language-specific code is usually much less complex, and fewer lines are needed per-language, as shown in Table 2.

5.3 Readability Study

Transforming through an IR mutilates the program, but transforming with incremental parametric syntax preserves information. To prove this, we ran a "Turing test," where our system and human programmers transformed code in the same manner, and judges from Mechanical Turk rated them

```
1  while (f() && g(1+1)) {
2      x++;
3  }
```

```
1  var t1 = f();
2  var t2;
3  if (t1) {
4    var t3 = 1 + 1;
5    t2 = g(t3);
6  }
7  var t4 = t1 && t2;
8  while (t4) {
9    x++;
10   t1 = f();
11   if (t1) {
12     var t3 = 1 + 1;
13     t2 = g(t3);
14   }
15   t4 = t1 && t2;
16 }
```

Fig. 19. TAC transformation example for JavaScript, showing handling of loops and non-strict operators.

```
1  if x is None:
2    doThing1()
3  elif x.foo():
4    doThing2()
```

```
1  t1 = x is None
2  if t1:
3      del t1
4      doThing1()
5  else:
6      del t1
7      t2 = x.foo()
8      if t2:
9          del t2
10         doThing2()
11     else:
12         del t2
```

Fig. 20. TAC transformation example for Python. It avoids computing `x.foo()` when `x` is `None`, and deletes all temporaries immediately after use, as Python is sensitive to the GC behavior. Adding the `del` statements is 5 lines of Python-specific code.

both on readability. This section presents highlights of this study; full details are given in Appendix Bof the arXiv version of this paper [Koppel et al. 2018]. Our study aimed to prove the following hypothesis for each language:

Hypothesis 1.
For a random code sample and our transformations, a human judge will rate the machine-transformed code at most 1 worse on a 1-5 scale than the human-transformed code in expectation.

Note that this study was completed using earlier versions of the transformations which failed some tests.

5.3.1 Phase 1: The RWUS Suite. As objects in our study, we needed (1) representative samples of real-world code, and (2) an objective measure of whether a transformed sample was equivalent to the original. As random samples of code do not come with thorough tests, we created our own.

The RWUS (Real World, Unchanged Semantics) suite consists of 50 functions across 5 languages randomly selected from top GitHub projects, together with a test suite designed to catch any semantic changes to the program. Each function is distributed as a file that can be compiled and executed without any dependencies. Our test cases have full path coverage and ensure all mocked

functions are called in the expected order with the expected arguments. The tests are incredibly thorough: while the actual samples total 1158 lines of code, the RWUS suite totals 8070 lines of code.

We expect the RWUS suite to be useful in testing other semantics-preserving transformations. It is available from:

https://github.com/jkoppel/rwus

5.3.2 Phase 2: Human-Written Transformations. In the next phase, we recruited programmers, gave them random entries from the RWUS suite, and had them perform each of our transformations by hand on the function. After normalizing formatting, 24 of the 120 resulting programs were identical to the machine-transformed ones. Another 7 of the machine-transformed ones failed their tests; the remaining programs were sent to human judges for Phase 3.

5.3.3 Phase 3: Mechanical Turk. In Phase 3, we asked human judges from Mechanical Turk to rate the manually-transformed code from Phase 2 along with their automatically transformed counterparts. In random order, they were given an original program and its human- and machine-transformed versions, and asked to rate both on a 1-5 scale, prioritizing correctness, then whether the transformation was done correctly, and third on code quality. We also employed measures to catch unqualified and inattentive Turkers and discard their data, chiefly canary questions (e.g.: a pair of identical programs which should be rated identically). We assigned each of the 20-30 transformed samples to 10 judges, giving us up to 300 ratings per language.

5.3.4 Results. For each language, we tabulated the difference in ratings between the human-written and automatically transformed programs. Our results are given in Figure 21. The average differences in ratings ranged from -0.075 for Python (favoring the humans) to $+0.633$ for Java (favoring the machine). The differences for C, JavaScript, and Lua were -0.014, $+0.396$, and -0.052 respectively.

We test Hypothesis 1 using the techniques of non-inferiority testing [Wellek 2010]. We combined the data from the human judges with the samples that did not get sent to Phase 3. For the samples where the human- and machine-transformed versions, were identical, we included them as if the humans had rated them identically; for the ones that failed their tests, as if rated to maximally penalize the machine. We then tested each of the 5 hypotheses using a paired t-test. For each language, it showed that the machine-transformed code was non-inferior by a non-inferiority margin of at most 1 with $p < 10^{-8}$. In retrospect, this data had the power to prove the hypothesis with a much smaller non-inferiority margin.

Considering both the raw data and the statistical tests, our study provides strong evidence that the output of transformations in Cubix is no less readable than hand-transformed code, showing that implementing source-to-source transformations with incremental parametric syntax avoids the mangling common to IR-based approaches.

5.4 Correctness

We claim it's feasible to write semantics-preserving language-parametric transformations with our approach. Hence, we collected language test suites for each of the 5 languages, and improved our transformations until we had a 100% pass rate for all transformations on all languages.

The caveat, though, is that there are some tests which the transformations should not pass. First, we use third-party parsers and pretty-printers, all of which have bugs. We contributed some bug fixes to all of these projects, but issues still remain. Second, all of the dynamic languages have self-referential tests which will never pass (e.g.: "assert this function was declared on line 37"). We rule out these cases by first checking if the test still passes after running the identity

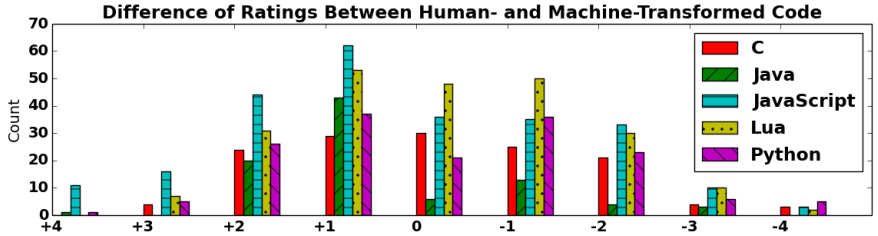

Fig. 21. Counts of differences between the ratings of the machine transformations and the human transformations. The leftmost bars represent cases where the judge rated the machine-produced output higher than the human-produced.

Table 3. Compilers/interpreters and test suites used in evaluation

Language	Compiler/Interpreter	Test Suite	Test Files	Total Test LOC
C	GCC 6.3.0_1	gcc-torture	1394	53,637
Java	JDK 1.8.0_65	K-Java	755	26,568
JS	Node.js v0.10.24	test262	2782	128,698
Lua	Lua.org 5.3.3	Lua Tests	28	12,017
Python	CPython 3.7.0a0	CPython Tests	404	249,499

transformation **Ident**, consisting of parsing and pretty-printing the program. 93.4% of tests pass this **Ident** transformation. This discussion excludes the Lua test suite, which has other issues explained below.

Table 3 lists the language implementations and test suites used in our evaluation. The C, Lua, and Python tests come from their implementations, while the JavaScript ones come from the official specification conformance test suite. The authors of K-Java, Bogdanas and Roşu [2015], report that no Java language tests are publicly available, and hence created their own specification tests, which we use. We restricted ourselves to the core language tests of `test262`, using the same subset as the JavaScript semantics KJS [Park et al. 2015], and omitted a small handful of multi-file Java tests among the Java ones, which caused problems with our test harness. We used the entirety of the Lua, Python, and C test suites.

Table 4 shows the number of passing tests for each language and transformation. The **Ident** transformation is a baseline transformation which simply parses and pretty prints a program, in order to filter out "bad tests" as described above. The **Hoist**, **Testcov**, and **TAC** columns show the results of their respective transformations. Comparing the other transformations to **Ident**, all but 12 tests pass. The failing JavaScript and Python tests are all self-referential tests that were not ruled out by **Ident**. The failing JavaScript tests all use `function . toString ()`, which retrieves the textual source code of the function. The Python ones inspect the runtime representation of functions, such as the presence of opcodes in the compiled bytecode or the number of bytes used per stack frame. The failing Java hoist test is actually due to a crash of `javac`. Manual inspection shows that this program is indeed correct, and the bug has been confirmed by the JDK developers [JDK Bug System 2016].

While we were very successful with the other language test suites, we found substantial barriers using the Lua test suite to test our transformations. Its tests are highly self-referential, including a check that the "test" function is defined on line 17, points where it undefines every global variable, and tests that break if a file changes character encoding. We nonetheless tried.

As the Lua tests are distributed as a single program, we modified the Lua test suite to maintain a count of passed assertions, instead of stopping at the first failure, and deleted some of the overly self-referential assertions. We found that the total number of calls to `assert` was nondeterministic,

Table 4. Results of each transformation on the test suites

Lang	Total	Ident	Hoist	Testcov	TAC
C	1394	1305	1305	1305	N/A
Java	755	745	*744	745	N/A
JS	2782	2573	2573	2568	2572
Python	404	360	N/A	358	357
Lua	Reported separately				
* Not including test which crashed `javac`					

but the number of failing assertions was not. In one set of runs, we obtained the following numbers: 70440/70456 passing assertions for the original, 70279/70295 for the identity transformation, and 70463/70479 for hoisting. We gave up attempting to get it working for the test coverage transform, due to crashes related to its metaprogramming around global variables. We similarly gave up for the TAC transformation, because the Lua VM does not allow for more than 200 local variables in any scope, and the TAC transformation overwhelms this easily. We conclude that the Lua test suite is unsuitable for testing program transformations.

6 RELATED WORK

Our work is most directly based on the data types à la carte approach to modular syntax [Swierstra 2008], and its extensions in work on compositional data types by Bahr and Hvitved [2011]. The extension to multi-sorted terms was introduced in Yakushev et al. [2009]. Other approaches to modular syntax include tagless-final [Kiselyov 2012], object algebras [Zhang et al. 2015], and modular reifiable matching [d. S. Oliveira et al. 2015]. All these works share the same limitation: supporting a language requires building it from scratch in terms of special components. We overcame this limitation by using sort-injections to intermix a generic representation with one from existing frontends. We previously described Cubix in a poster-paper in OOPSLA 2017 [Koppel and Solar-Lezama 2017].

This work on modular syntax is joined by work on modular semantics, such as modular monadic semantics [Liang et al. 1995] and its cousin modular monadic meta-theory [Delaware et al. 2013], as well as modular SOS [Mosses 2004] and its successor work on funcons [Churchill et al. 2015]. These are used to build and verify interpreters for multiple languages, and will likely be necessary to extend our work to verifying multi-language transformations.

2003 saw a Dutch grant on language-parametric refactoring [van de Brand et al. 2003], building on a prototype by Lämmel [Heering and Lämmel 2004; Lämmel 2002]. Their approach was to parameterize a transformation on (1) a fixed number of (language-specific) sorts used by the transformation, and (2) a set of primitive transformation operations, given as functions over these sorts. In this approach, the ASTs are opaque to the generic code, and hence essentially all computation happens in the language-specific functions. Conversely, in our approach, the generic code can manipulate the generic portions of a tree directly, which allows large chunks of a language-parametric transformation to be written similarly to a normal single-language rewrite.

Sort injections are an instance of the concept of *feature interactions* from the field of software product lines [Van Gurp et al. 2001]. A similar idea is seen in the TruffleVM [Grimmer et al. 2015] to allow language runtimes to exchange messages.

The past decade has seen extensive work in *language workbenches*, which are designed to make it easy to implement languages and transformations on them. They include Spoofax and its component Stratego [Kats and Visser 2010], Rascal [Klint et al. 2009], TXL [Cordy 2006], Semantic Designs DMS [Baxter et al. 2004], and JetBrains MPS [Voelter and Pech 2012]. These were extensively surveyed

in Erdweg et al [Erdweg et al. 2013]. All these share the limitation that, while they make it easy to define languages and write transformations, the resulting transformations can only run on one representation of one language. At best they can be used to implement the "Clang-style" common representation, discussed in Section 1.1.

One recent work that echoes our own is Brown et al's [Brown et al. 2016] work using *island grammars* [Moonen 2001] to write static analyzers for multiple languages. They show that they only need to represent fragments of a language to construct an analyzer. Their analyzers are still built for a single language, and they resort to cloning code to implement them for others. They do not address transformation.

Incremental concrete syntax [Dinkelaker et al. 2013] is a technique using island grammars to construct parsers. It focuses on concrete syntax (parsing); ours is on abstract syntax (representation).

7 CONCLUSION

Incremental parametric syntax fulfills a simple promise: when writing similar transformations for multiple languages, they should be able to share code to the extent the languages are similar. We think that the ability to make multi-language transformation tools will greatly increase the cost/benefit ratio of building tools, and other researchers are noticing. In our previous presentations of Cubix, we were approached by groups from Microsoft Research and Uber who wished to implement our approach to support their own multi-language tooling, while several other groups inquired about using Cubix itself in their research. One from the University of Washington has already begun doing so. We plan to work extensively on supporting these efforts after the public release of Cubix.

The work in this paper is, to our knowledge, the first to allow a single program to perform source-to-source transformations on multiple real languages while preserving the information of each. We believe incremental parametric syntax solves a key problem in writing multi-language tools. The Cubix framework is available from:

https://github.com/jkoppel/cubix

The RWUS suite is available from:

https://github.com/jkoppel/rwus

ACKNOWLEDGMENTS

We thank Chris Barnett and Jonathan Paulson for assisting in the construction of the RWUS test suite, and to Dieter Vekeman for helping to fix pretty-printer bugs in our dependencies. We thank Carrie Cai, Jiasi Shen, and Ethan Bian for advising in the experimental and visual design of the study. We further thank our other Cubix contributors, Jasper Haag and Máté Kovács. Finally, we thank everyone who has given us feedback on earlier drafts of the paper, especially the anonymous reviewers, and the many audiences who have given us feedback on presentations of Cubix.

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1122374.

REFERENCES

Franz Baader and Tobias Nipkow. 1999. *Term Rewriting and All That.* Cambridge university press.

Patrick Bahr and Tom Hvitved. 2011. Compositional Data Types. In *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic programming, WGP@ICFP 2011, Tokyo, Japan, September 19-21, 2011.* 83–94.

Ira D Baxter, Christopher Pidgeon, and Michael Mehlich. 2004. DMS®: Program Transformations for Practical Scalable Software Evolution. In *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, 625–634.

Denis Bogdanas and Grigore Roşu. 2015. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 445–456.

Niklas Broberg. 2015. language-java: Manipulating Java source: abstract syntax, lexer, parser, and pretty-printer. http://hackage.haskell.org/package/language-java-0.2.8. (November 2015).

Fraser Brown, Andres Nötzli, and Dawson Engler. 2016. How to Build Static Checking Systems Using Orders of Magnitude Less Code. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 143–157.

David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *CAV*.

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

Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer, and José F Quesada. 2002. Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science* 285, 2 (2002), 187–243.

James R Cordy. 2006. The TXL Source Transformation Language. *Science of Computer Programming* 61, 3 (2006), 190–210.

Bruno C. d. S. Oliveira, Shin-Cheng Mu, and Shu-Hung You. 2015. Modular Reifiable Matching: A List-of-Functors Approach to Two-level Types. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*. 82–93.

Benjamin Delaware, Steven Keuchel, Tom Schrijvers, and Bruno C. d. S. Oliveira. 2013. Modular Monadic Meta-Theory. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. 319–330.

Isil Dillig, Thomas Dillig, and Alex Aiken. 2009. SAIL: Static Analysis Intermediate Language with a Two-level Representation. *Stanford University Technical Report* (2009).

Tom Dinkelaker, Michael Eichberg, and Mira Mezini. 2013. Incremental Concrete Syntax for Embedded Languages with Support for Separate Compilation. *Science of Computer Programming* 78, 6 (2013), 615–632.

Michael Dory, Allison Parrish, and Brendan Berg. 2012. *Introduction to Tornado: Modern Web Applications with Python*. O'Reilly Media, Inc.

Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. 2013. The State of the Art in Language Workbenches. In *International Conference on Software Language Engineering*. Springer, 197–217.

Python Software Foundation. 2016. CPython Test Suite. Version 3.7.0a0. https://docs.python.org/devguide/runtests.html. (October 2016).

FSF. 2016. C Language Testsuites: "C-torture". Revision 240758. http://gcc.gnu.org/onlinedocs/gccint/C-Tests.html. (October 2016).

Andy Gill. 2009. A Haskell Hosted DSL for Writing Transformation Systems. In *Domain-Specific Languages*. Springer, 285–309.

Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. 2015. High-performance cross-language interoperability in a multi-language runtime. In *Proceedings of the 11th Symposium on Dynamic Languages, DLS 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. 78–90.

Miguel Grinberg. 2014. *Flask Web Development: Developing Web Applications with Python*. O'Reilly Media, Inc.

Jan Heering and Ralf Lämmel. 2004. Generic Software Transformations. In *Proceedings of the Software Transformation Systems Workshop*.

Benedikt Huber. 2016. language-c: Analysis and generation of C code. http://hackage.haskell.org/package/language-c. (2016).

JDK Bug System. 2016. javac crash when local from enclosing context is captured multiple times. https://bugs.openjdk.java.net/browse/JDK-8169345. (2016).

Lennart CL Kats and Eelco Visser. 2010. *The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs*. Vol. 45. ACM.

Oleg Kiselyov. 2012. Typed Tagless Final Interpreters. In *Generic and Indexed Programming*. Springer, 130–174.

Paul Klint, Tijs van der Storm, and Jurgen Vinju. 2009. EASY Meta-programming with Rascal. In *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer, 222–289.

James Koppel, Varot Premtoon, and Armando Solar-Lezama. 2018. One Tool, Many Languages: Language-Parametric Transformation with Incremental Parametric Syntax. *CoRR* abs/1707.04600 (2018). arXiv:1707.04600 http://arxiv.org/abs/1707.04600

James Koppel and Armando Solar-Lezama. 2017. Incremental parametric syntax for multi-language transformation. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, SPLASH 2017, Vancouver, BC, Canada, October 23 - 27, 2017*. 53–54. https://doi.org/10.1145/3135932.3135940

Ralf Lämmel. 2002. Towards Generic Refactoring. In *Proceedings of the 2002 ACM SIGPLAN workshop on Rule-based programming*. ACM, 15–28.

Ralf Lämmel and Simon Peyton Jones. 2003. *Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming*. Vol. 38. ACM.

Ralf Lämmel, Eelco Visser, and Joost Visser. 2003. Strategic Programming Meets Adaptive Programming. In *Proceedings of the 2Nd International Conference on Aspect-oriented Software Development (AOSD '03)*. ACM, New York, NY, USA, 168–177. https://doi.org/10.1145/643603.643621

Ralf Lämmel and Joost Visser. 2002. Typed Combinators for Generic Traversal. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 137–154.

Ralf Lämmel and Joost Visser. 2003. A Strafunski Application Letter. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 357–375.

Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. 75–88.

Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 333–343.

Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java Virtual Machine Specification: Java SE 8 Edition*. Pearson Education.

Sam Madden. 2017. 6.830 Lab 1: SimpleDB. (2017). http://db.csail.mit.edu/6.830/assignments/lab1.html

Leon Moonen. 2001. Generating Robust Parsers using Island Grammars. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*. IEEE, 13–22.

Peter D. Mosses. 2004. Modular Structural Operational Semantics. *J. Log. Algebr. Program.* 60-61 (2004), 195–228.

Ömer Sinan Ağacan and Eric Mertens. 2016. language-lua: Lua parser and pretty-printer. http://hackage.haskell.org/package/language-lua-0.10.0. (August 2016).

Daejun Park, Andrei Stefănescu, and Grigore Roşu. 2015. KJS: A Complete Formal Semantics of JavaScript. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 346–356.

Bernard James Pope. 2016. language-python: Parsing and pretty printing of Python code. http://hackage.haskell.org/package/language-python-0.5.4. (July 2016).

PUC-Rio. 2016. Lua: Test suites. Version 5.3.3. https://www.lua.org/tests/. (2016).

Semantic Designs, Inc. 2005. Test Coverage tools. http://www.semanticdesigns.com/Products/TestCoverage/. (2005).

Tim Sheard and Simon Peyton Jones. 2002. Template Meta-Programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. ACM, 1–16.

Wouter Swierstra. 2008. Data Types à la Carte. *Journal of Functional Programming* 18, 04 (2008), 423–436.

ECMA TC39. 2014. Test262: ECMAScript Language Conformance Test Suite. Version 5.1. http://test262.ecmascript.org. (2014).

Danny van Bruggen. 2016. JavaParser: Process Java code programmatically. (2016). http://javaparser.org

Mark van de Brand, Jan Heering, Paul Klint, Ralf Lämmel, and Christian Verhoef. 2003. Language-Parametric Program Restructuring. (2003). http://www.cs.vu.nl/lppr/abstract/abstract.html

Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. 2001. On the Notion of Variability in Software Product Lines. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*. IEEE, 45–54.

Markus Voelter and Vaclav Pech. 2012. Language Modularity with the MPS Language Workbench. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 1449–1450.

Philip Wadler. 1987. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 307–313.

Philip Wadler. 1998. The Expression Problem. *Java-genericity mailing list* (1998).

Stefan Wellek. 2010. *Testing Statistical Hypotheses of Equivalence and Noninferiority*. CRC Press.

Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. 2009. Generic Programming with Fixed Points for Mutually Recursive Datatypes. In *Proceeding of the 14th ACM SIGPLAN International Conference on Functional Programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*. 233–244.

Haoyuan Zhang, Zewei Chu, Bruno C. d. S. Oliveira, and Tijs van der Storm. 2015. Scrap Your Boilerplate with Object Algebras. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. 127–146.

Alan Zimmerman. 2016. language-javascript: Parser for JavaScript. http://hackage.haskell.org/package/language-javascript-0.6.0.9. (November 2016).