Incremental Parametric Syntax for Multi-Language Transformation

by

James Koppel

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2017

Author .....
Department of Electrical Engineering and Computer Science
January 31, 2017

Certified by .....
Armando Solar-Lezama
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Professor Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

Incremental Parametric Syntax for Multi-Language Transformation

by

James Koppel

Submitted to the Department of Electrical Engineering and Computer Science
on January 31, 2017, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

We present a new system 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 construct incremental parametric syntaxes for C, Java, JavaScript, Lua, and Python, and demonstrate two multi-language program transformations that can run on all of them. Our evaluation shows that (1) once a transformation is written, relatively 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) despite dealing with many languages, our transformations can still handle language corner-cases, and pass 90% of compiler test suites.

Incremental parametric syntax is based on the datatypes à la carte approach for constructing modular syntax, but extends it with the notion of a **sort injection**, which allows intermixing language-specific and generic components in a type-safe and modular fashion. Instead of translating each language to a common representation, this allows having a family of representations, each specific to a language, but sharing common components. Our system can construct an incremental parametric syntax semi-automatically from a third-party library for that language, with the user only writing code for the portions they wish to translate into generic components. The resulting incremental parametric syntax is isomorphic to the original representation, allowing transformations to be fully information-preserving. The user can begin by only translating a small fragment of a language into generic components, enough to support a few transformations, and incrementally add more. Our experience shows that constructing an incremental parametric syntax for a new language is easy, typically taking less than a day of work, and that multi-language transformations built against incremental parametric syntaxes can be configured for a large number of languages, with only a small amount of work per language.

Thesis Supervisor: Armando Solar-Lezama

Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank my adviser Armando Solar-Lezama for his guidance and unwavering support through times both good and bad. Without his insight and perspective, the ideas in this thesis may have been left as a sidenote on another project.

Thanks to 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. And thanks to the many people who helped me in both the visual and experimental design of the study, including Ethan Bian, Carrie Cai, and Jiasi Shen

I did the initial work in this thesis back in 2013 when I was founder of Tarski Technologies. I would like to thank Peter Thiel and the Thiel Foundation for funding me during that time, and I would like to thank Alex Aiken for the conversation in 2012 that sparked my interest in the multi-language problem.

And finally, I would like to thank my parents Ralph and Suellen, and my sister Julie, for their twenty-some years of unconditional love.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the scale of software grows, developers will increasingly depend on program transformation tools to help maintain software. Programmers use transformation tools to do everything from small-scale refactoring [15] to modernizing entire legacy applications [1]. Most tools are wedded to one language (or even one compiler), and often require hundreds of thousands of lines of code to implement [6]. Consequently, as developers use hundreds of languages [22], each tool has a limited potential market, and hence very few are built.

To solve this problem, implementers need to be able to share code when writing similar tools for different languages. In this direction, researchers in the past few decades have made great progress, developing techniques for dealing with different languages modularly, including work on modular semantics [9, 28], modular interpreters [25], and modular syntax [2, 42]. Using any of these techniques requires defining an entire language in a specialized manner, differently from existing tools. Hence, so far they have only been applied to small languages such as DSLs.

In this paper, we present **incremental parametric syntax**, which enables implementers to define languages modularly on a much greater scale than previously, and hence write source-to-source transformation tools that run on multiple real programming languages. We demonstrate this by developing several characteristic program transformations that each can run on several of C, Java, JavaScript, Lua, and Python. We show that users can define these transformations in a few lines of code. In our

study, users rated the output of our transformations as equal or preferable to that of modifying the code by hand, modulo formatting differences.

Although many previous tools exist for writing analyses or even optimizing code-generators for multiple languages, their design fundamentally prevents them from expressing source-to-source transformations. The common pattern when implementing tools for multiple languages is to translate programs in each source language into some abstracted *intermediate representation*, and then to implement the analysis on that IR. This pattern is realized by many program analysis frameworks such as SAIL [11], CIL [29], and BAP [7]; as well as bytecode VMs such as LLVM [23], the JVM [26], and the .NET CLR. Conceptually, this consists of providing the following functions:

```
lower   :: Java →IR, C →IR, Python →IR, ...
analyze :: IR →AnalysisResult
```

The problem becomes much harder when a user wishes to not only analyze but also transform programs. The natural extension of the above model is to define transformations on the IR, and then further provide a *lift* function from the IR to the source language:

```
transform :: IR →IR
lift      :: IR →Java, IR →C, IR →Python, ...
```

By composing it with the `lower` and `lift` functions, this allows a programmer to turn a single generic transform `transform` ::`IR` →`IR` into multiple language-specific transforms $\texttt{transform}_J$::`Java` →`Java`, $\texttt{transform}_C$::`C` →`C`, etc. A close reader can already see that this approach is fundamentally limited: note that one can compose the `lower` and `lift` functions to get a translation from any language to any other!

Roughly speaking, there are two ways to construct this IR. One is to normalize everything to lower-level representation, i.e.: the least-common-denominator of all the languages. This is the approach taken by most compilers, as well as bytecodes such as LLVM. Doing this will necessarily destroy information. As a common example, if the IR abstracts multiple kinds of loops into `while` loops, then any source-to-source

transform which passes through the IR will convert all loops into `while` loops.

The other approach, seen in frameworks such as Clang and WALA, is to embed the *union* of all concepts in the languages into the IR, putting language-specific nodes into the "common" representation. 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 also allows the representation of invalid programs (by, e.g.: freely mixing Objective-C and C++ concepts), and necessitates that the `lift` functions be partial.

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 human consumption.

Conversely, rather than providing a single common representation for multiple languages, our approach instead provides a family of representations with shared components, but with each representation specialized to a single language. An *incremental parametric syntax* is written as a modification to an existing syntax definition. It produces a representation of a language as a composition of generic fragments, and a subset of the original, language-specific definition. This representation is isomorphic to the original, and hence preserves all information. The user then writes transformations on the generic fragments and lifts them to the rest of language, allowing them to be as precise as transforms written specifically for each language. As the user desires an increased scope of transformation, they can simply re-express more of an existing language using parametric syntax. Incremental parametric syntax, combined with our translation infrastructure, allow for reusing existing, independently-developed frontends for each language. As a result, supporting a new language requires relatively little work.

Conceptually, one can think of our approach as offering the following functions. Incremental parametric syntax provides the ability to decompose a language into generic and specific parts, and define transformations which run on any representation containing the specific parts.

```
decompose_J :: Java →IR ⋈Rem_J
decompose_C :: C →IR ⋈Rem_C
transform   :: ∀x. IR ⋈x →IR ⋈x
recompose_J :: IR ⋈Rem_J →Java
recompose_C :: IR ⋈Rem_C →C
```

It should be noted that, in our approach, our code generation tools automatically derive the $\texttt{Rem}_J$ fragment the original `Java` fragment with a small amount of specification, and likewise for $\texttt{Rem}_C$. The user can also incrementally shift more of a language from the language-specific to the generic fragments in order to support new transformations. In our experience, adding support for a new language takes less than a day of work. Much of this time is spent on the inescapable task of matching the semantics of the language with the semantics of the generic components.

The $X \bowtie Y$ notation represents a manner of composing two syntaxes known in the term-rewriting community as "sum of signatures" and known in the functional-programming community as "data types à la carte" [35]. Our key addition is the notion of a **sort injection**, which provides a mechanism for specifying the interactions between the different components. A sort injection is a node or sequence of nodes which allows terms of one sort to be used at another sort. For instance, they allow a user to specify that a generic assignment may be used where a language-specific statement was previously expected, and to independently specify what terms may appear as the left-hand-side and right-hand-side of an assignment. Using these sort injections, our system can generate code for a new representation of the language which is isomorphic to the original one, and which allows intermixing language-specific and generic fragments.

Knowing that a language contains a generic component implies that it supports certain operations, and similarly for when it has a sort injection. These each form a node-level interface on the language's syntax trees, so a program can e.g.: take a node of an unknown language that contains assignments, and query whether that node is an assignment node. Users then define transformations parameterized on these operations, so that they are applicable to any language which provides the required

interfaces. To those, we add a handful of other interfaces providing primitives such as statement insertion. These are sufficient to implement a rich library for multi-language program transformation and control-flow graph generation, which we use to build sophisticated transformations.

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 a **sort injection**, which provides a typed and modular way to intermix language-specific and generic fragments.
- We present an algorithm for automatically converting a datatype into the sum-of-products representation, and present `comptrans`, a code-generation library for doing so.

We use these techniques, along with existing ones, to produce the following results.

- We provide a demonstration 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 complicated transformations based on control-flow, yet still achieve results that are as readable as hand transformed code
- 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 any modification that changes program semantics.
- We present the results of a human study comparing the output of our transformations against hand-transformed code. Our transformations were identical

to the hand-transformed code over 25% of the time, and, of the rest, human judges gave ours higher average scores for readability

Chapter 2

Overview

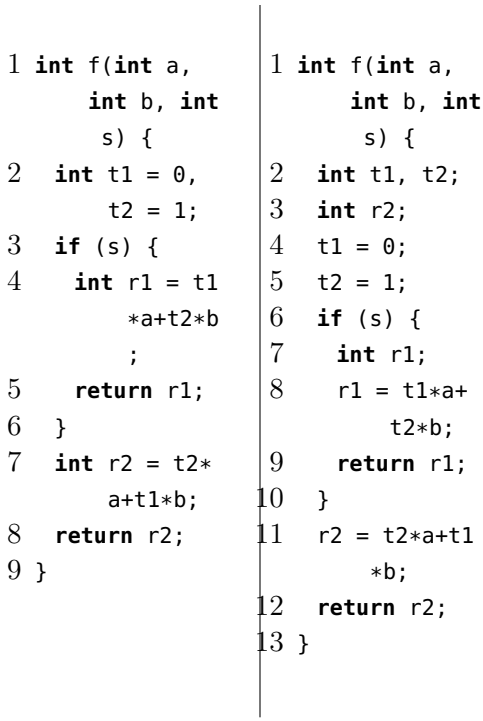
In this section, we show how our system allows constructing multi-language transformations, and the work required to add support for a new language. In Section 2.1, we explain the construction of a transformation called "declaration hoisting," and how it is configured to run on several languages. Section 2.2 then gives a detailed explanation of creating an incremental parametric syntax for C.

2.1 Building a Hoisting Transformation

In this section, we describe the construction of a transformation for *declaration hoisting*. We show how, with a small amount of configuration, we can apply this transformation to C, Java, JavaScript, and Lua. We do not apply this transformation to Python, because Python lacks variable declarations.

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 be written. If a variable of the same name is referenced earlier in the scope, we do not hoist. Figure 2-1 gives an example C program and its hoisted version.

To build a multi-language transformation, the user declaratively writes down the set of abstract language constructs that need to be supported, and the transformation will be applicable to any language that satisfies them. We refer to this set as



```
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;
3   int r2;
4   t1 = 0;
5   t2 = 1;
6   if (s) {
7     int r1;
8     r1 = t1*a+
          t2*b;
9     return r1;
10  }
11  r2 = t2*a+t1
        *b;
12  return r2;
13 }
```

Figure 2-1: An example of hoisting a C program

constraints, and explain in section 4 how this is realized. In the case of hoisting, the primary constraints are that language must contain variable declarations, assignments, blocks, and identifiers. Additionally, both assignments and variable declarations must be valid members of blocks. The latter is expressed using our notion of *sort injections*, described in Section 2.2.

To implement hoisting, the user defines a transformation over blocks, and then uses our higher-order tree traversal library to lift this into a bottom-up rewrite of the entire program. The transformation checks if each block item is a variable declaration. If so, it splits the variable declaration into a variable declaration without initialization, and 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. The transformation also tracks the set of referenced identifiers so it can avoid hoisting declarations past uses of variables with the same name. This implementation is overall quite conservative, and errs on the side of not hoisting.

While the basic implementation is simple, like any program transformation, much of the work comes in dealing with language subtleties— and, in this case, with variation between languages. We'll explain briefly how our implementation deals with a few of these.

Two major relevant points of variation are whether a language allows multiple variables to be declared simultaneously, and whether a variable initializer can always be used as the right-hand-side of an assignment. Our implementation solves the first problem using a *d*isjunctive constraint. It requires that the language support either single or multi variable declarations, and dispatches appropriately at the relevant point. For the latter problem, our implementation adds a new constraint demanding that the user write an operation for each language to convert from variable declarations to assignment right-hand-sides. For Lua and JavaScript, this translation is trivial. For Java and C, which have different syntax for initializing an array versus assigning an array, some implementation is required. Implementing them becomes the main way that this transformation must be configured per language.

Transformations implemented in our system maintain the flexibility to deal with language-specific edge-cases. Often, no special code is needed in the transformation at all, and the problem can instead be dealt with by enhancing the representation. For an example of the former, in JavaScript directives such as `''use strict";` must be placed at the top of the block to have effect, meaning that hoisting a declaration above it can break the code. Because these directives do not directly take part in computation, and are in some ways treated in the JavaScript specification as a separate kind of syntax, we decided that they should not be considered part of the block, and modified the representation of JavaScript to store them separately. The hoisting transformation immediately started handling this special case gracefully, and this also fixed a similar bug in another transformation.

Despite dealing with the complexity of four languages, our implementation contains only 80 lines of Haskell for the core transformation, plus an average of 15 lines of configuration code per language. It depends on some more general configuration for dealing with variation in whether a language can declare multiple variables simultane-

ously, totalling 30 lines of code, as well as the sort injections for each language from assignments and variable declarations to block items. The type class declarations corresponding to the sort injections average 12 lines of code per language.

2.2 Modularizing C

In Section 2.1, we outlined how to build a multi-language transformation, which works on any language that contains some common notion of variable declarations and a few other constructs. In this section, we give the detailed example of constructing an incremental parametric syntax for C, in which parts of of the language are recast in terms of these common components, so that mullti-language transformations can be applied.

To support a language, our system requires that it have three representations. The final form is the incremental parametric syntax mentioned above. The starting point is some already-existing syntax definition of the language from a 3rd-party library, with its accompanying parser and pretty-printer. In between is the "modularized" representation. The modularized representation is derived automatically from the original representation, but splits the language into many components. This enables the user to define the final incremental-parametric representation by replacing some components of the modularized representation with their generic equivalents. These three representations are mutually isomorphic. Translations between the original and modularized definitions are derived completely automatically, while translations between the modularized and incremental-parametric representations are generated mostly-automatically, with the user only needing to write code for the points where they differ, i.e.: for the language-specific components which have been replaced with generic ones. Figure 2-2 depicts the process of creating these three representations, while Figure 2-3 shows the path of a program at runtime.

In the remainder of this section, we demonstrate how to construct an incremental parametric syntax for C. For reasons that will soon be clear, our example uses the Haskell library `language-c` [17] as a starting point.

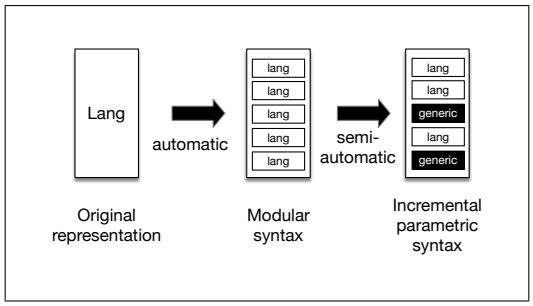


Figure 2-2: Each language requires three representations in our approach. The incremental parametric syntax is constructed from the modularized syntax, which is in turn automatically derived from the original representation, taken from a 3rd-party library.

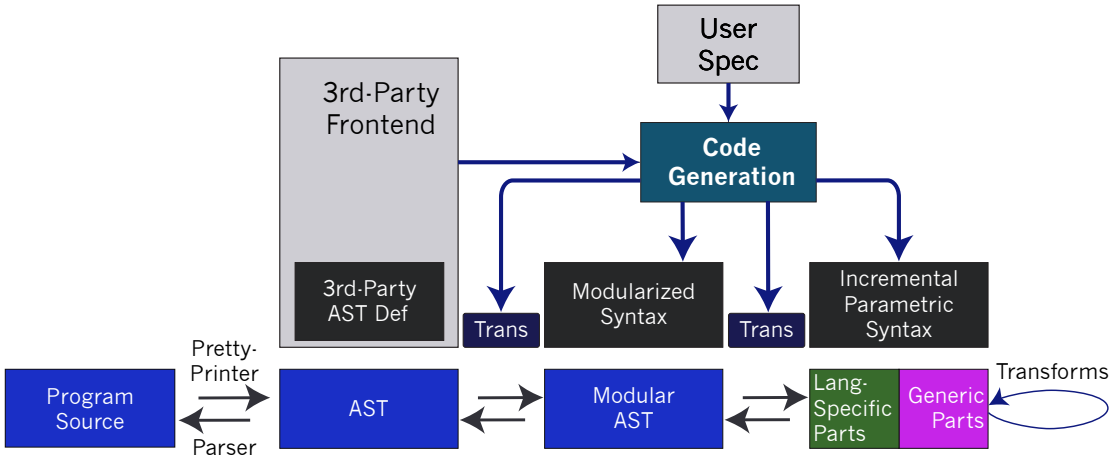


Figure 2-3: At runtime, a program is progressively transformed through each representation, until it is decomposed into generic and language-specific components, so that a multi-language transformation can be applied. To render a program back into source, this process is then run in reverse.

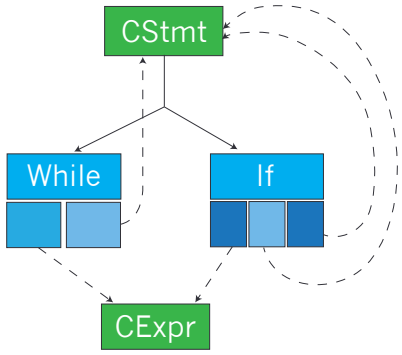


Figure 2-4: A fragment of a typical representation of C. The self-reference hinders modularity.

Figure 2-4 depicts a fragment of the `language-c`'s datatype for C programs. The main barrier towards writing multi-language transformations is the recursion in the datatype. Consider a hypothetical transformation that swaps the two branches of an if-statement. A user might like to define this in a way that it can work on if-statements in any language. However, with this kind of representation, any two languages it operates on must use the same representation of statements.

The goal of the modularized representation is to solve this problem by removing the recursion from ("unfixing") the datatype, and then splitting the resulting unfixed language into fragments. The left column of Figure 2-5 depicts one such fragment, while the right column shows how they are combined back into a representation of the whole language. The modularized representation has an independent component for each sort of the original representation. In each component, all recursive occurences of the datatype have been replaced with a type variable, allowing the system to later specify what is allowed as a subterm of each node. These fragments are signatures that define the shape of each node in C. They are combined to give a signature for all allowed nodes in C. The final step in constructing the modularized representation is to close the recursion by applying a type-level fixpoint, declaring that the relevant sort of C node may be used anywhere where a type variable is indicated, thus obtaining a representation isomorphic to the original. This is the sum-of-products representation, or datatypes à la carte [35].

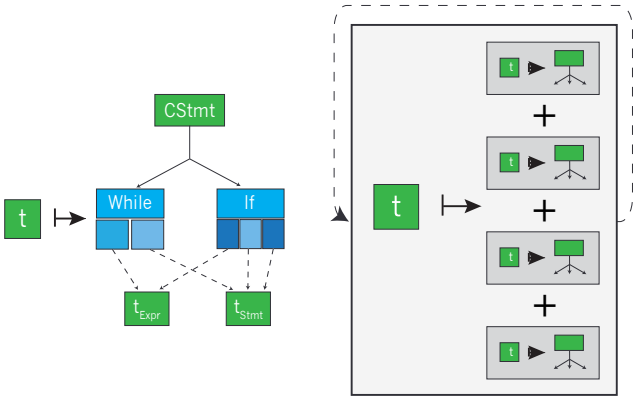


Figure 2-5: In the sum-of-signatures representation, a language is represented by a list of subsignatures like the one on the left. Each signature has a type variable for subterms, in lieu of self-reference. The subsignatures can be combined into a signature for the whole language, which is then closed by specifying that allowed subterms of terms of this signature are other terms of this signature (right). In this representation, it is easy to add, remove, and replace the subsignatures of the language.

We have implemented the modularization transformation in a Template Haskell library called `comptrans`, which takes an AST definition as a Haskell algebraic data type, and generates code for the modularized representation, along with translations to and from the original. We describe `comptrans` and the modularization process in more detail in Section 3.2. The user simply writes a small amount of boilerplate to invoke `comptrans` on `language-c`, and the modularized representation is generated automatically, along with accompanying translations to and from the `language-c` representation.

Because the modularized representation is composed of a list of signatures, it is easy to construct a new representation by adding and removing items from this list. In doing so, the user can slowly swap out C-specific parts for generic ones, gradually creating a parametric syntax for C. This process is depicted in Figure 2-2. The user initially only needs to genericize enough of the language to support whatever analyses and transformations are currently desired, and can incrementally replace more of the language with generic components as they wish to add more transformations. This is the incremental parametric syntax.

In this case, in order to support the hoisting transformation, the user must replace

C assignments, variable declarations, blocks, and identifiers with their corresponding generic components. Each of these generic components comes with an informal semantics. The user must first check these semantics against the C specification to ensure that the generic components properly model the relevant portion of C.

The user must next add nodes which determine the interactions between the C-specific and generic components. The user adds nodes indicating that assignments are C expressions, and that C expressions form the left-hand and right-hand sides of an assignment. They must also add nodes indicating that generic blocks may be used where a C block is expected, and that variable declarations and C statements form the entries of a block. These nodes establish **sort injections** between the generic and C-specific sorts, controlling what may be used where. Because the hoisting transformation requires a sort injection from variable declarations and assignments to block item, the user also writes code for a couple of derived sort injections. For instance, the sort injection from assignments to block items is a chain of nodes: assignments are expressions, expressions may be used as statements in C, and statements are block items.

The incremental-parametric signature for C can now be defined by taking the list of signatures automatically constructed from the `language-c` datatype, adding the signatures for the generic components and sort injection nodes, and subtracting the now-redundant signatures for C-specific identifiers and blocks. If the user were to replace the other C expressions with generic components, or define a new type for all C expressions except assignments, they could also remove the existing CExpression type from our signature. The final incremental parametric syntax is obtained by taking a type-level fixpoint of this signature, indicating that the children of each node of this signature are other nodes of this signature. This yields the end result: a modularized representation of C, isomorphic to the original, but with generic components for assignments, blocks, variable declarations, and identifiers.

The final step is to declare the correspondence between the constructs that have been removed from the language, and the generic components that have replaced them. Our system then uses these to construct the translations between the modular-

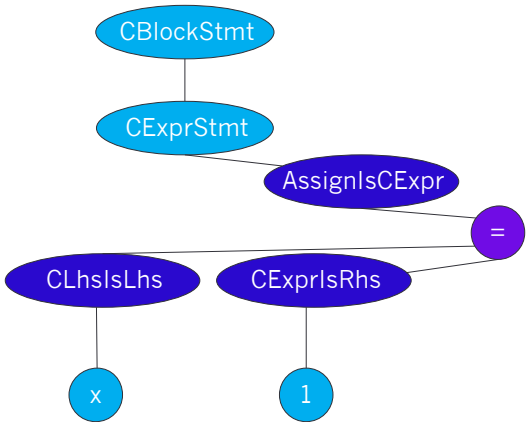


Figure 2-6: Portion of an example term in the modular representation of C, showing the intermixing of language-specific and generic components, with sort injection nodes at the boundary.

ized and incremental-parametric representations. Because these two representations are both modular, and mostly the same, little code is required: Our actual implementation of these translations for C totals 130 lines of Haskell code, about 40 of which are boilerplate.

Chapter 3

Core Ideas

In this section, we explain the core new ideas that makes our multi-language transformations possible. Section 3.1 presents the basic concepts of incremental parametric syntax. Section 3.3 presents sort injections. Section 3.2 explains the translation of a syntax into its modularized version.

3.1 Incremental Parametric Syntax

Consider sorting a list. By parameterizing out the `compare` function, one can write a single `sort` function which can operate on lists of any datatype, and yet be equivalent to having a separate `sort` function for each type. With some basic compiler optimizations, it can even be as efficient. It is this exact combination of generality and specialization which we aim to get for program transformations using parametric syntax.

Definition 1 *A **parametric syntax** defines a language as a composition of fragments, in a manner that allows operations to be defined on all languages which contain a given fragment.*

We have already presented a function over parametric syntax, the function `transform` $::\forall$`x. IR`$\bowtie$`x` $\rightarrow$`IR`$\bowtie$`x` from Section 1. We can now give more detail about the exact form of these types: the hoisting transformation from Section 2.1 has a type

similar to

`hoist` $::\forall$ `f. (Block` $\prec$`f, VarDecl` $\prec$`f)` $\Rightarrow$`f` $\rightarrow$`f`, where `x` $\prec$`f` indicates that signature f contains a fragment x. The incremental parametric syntax for C developed in Section 2.2 is a parametric syntax, and the `hoist` function can be applied to it. We will present the actual types later, after discussing the distinction between signatures and terms.

Strictly speaking, the modularized version of

`language-c` from Section 2.2 is already a fully parametric syntax — the representation makes it possible to e.g.: write a transformation over any language which contains the `CStatement` fragment. Since this gives no advantage over writing a transformation specific to C, we henceforth use the term "parametric syntax" only to refer to syntax fragments which may usefully be shared between languages, and a **fully parametric syntax** to refer to a representation where all syntax fragments are generic.

Our notion of parametric syntax is closely related to the Expression Problem [40], which concerns being able to separately extend a language with new terms and new operations. Any parametric syntax must also be a solution to the Expression Problem, but a solution to the Expression Problem need not allow for expressing multiple languages.

Developing a fully parametric syntax for a programming language can be extremely difficult. With an incremental parametric syntax, a comparatively small initial effort is needed.

Definition 2 *An **incremental parametric syntax** expresses a language as a hybrid between language-specific and parametric syntax, in a manner which allows the language-specific components to be converted piecewise into parametric ones.*

For an example, see the incremental parametric syntax for C that we described in Section 2.2.

3.2 Generating the Modularized Representation

Our `comptrans` inputs a syntax definition as system of mutually recursive algebraic datatypes, and outputs code for a new syntax definition in the sum-of-signatures

format isomorphic to the original. The latter is represented as a compositional data type [2]. Figure 3-1 gives an example of this transformation.

The transformation is extremely straightforward. For each inputted algebraic datatypes of kind $*$, a GADT of kind $(*\to*)\to(*\to*)$is created to represent modular terms, and a "label" type of kind $*$ is created to represent the corresponding sort. The first argument is the type of subterms, similar to in Figure 2-5. The second argument is a label indicating the sort of the term. These higher-order functors are summed together into a signature for a language, and then closed using the type-level fixpoint operator of kind $(*\to*)\to(*\to*)$, as shown in Figure 3-3. The GADT accepts one parameter for each parameter of the input ADT, but, instead of directly referencing the type of the subterm, it instead uses the type parameter.

This modularized representation . Any summand of a signature may be removed

The modularized representation has advantages over the original mutually-recursive ADT representation, even when writing transformations for only one dialect of one language. For instance, this representation allows giving a type `Term Sig l` $\to$`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 [2] for a more thorough discussion of this representation.

3.3 Sort Injections

While the sum-of-signatures representation modularizes what nodes can be in an AST, sort injections modularize what edges can be in an AST. The idea of a sort injection is simple. However, they are extremely powerful.

Definition 3 *A **sort injection** is a constraint stating that a term of one sort may be used at another sort.*

Definition 4 *A **sort injection node** is an unary production in a syntax definition added for the purpose of enabling a sort injection*

```haskell
data Arith = Add Atom Atom
data Atom = Var String | Const Lit
data Lit = Lit Int
```

```haskell
data Arith e l where
  Add :: e AtomL -> e AtomL -> Arith e
      ArithL

data Atom e l where
  Var :: String -> Atom e AtomL
  Const :: e LitL -> Atom e AtomL

data Lit (e :: * -> *) l where
  Lit :: Int -> Lit e LitL
```

Figure 3-1: `comptrans` takes a syntax definition as an ADT like the example on the left, and produces a modularized representation as a set of GADT definitions like on the right. The resulting GADTs can be combined using sum and fixpoint operators to obtain a representation isomorphic to the input.

```haskell
data (:+:) f g e l = Inl (f e l) | Inr (g e l)
data Term f l = Term (f (Term f)) l

type LangSig = Arith :+: Atom :+: Lit
type LangTerm = Term LangSig
```

Figure 3-2: Combining the fragments produced by `comptrans` in Figure 3-1

```
1 data AssignIsCExpr e l where
2   AssignIsCExpr :: e AssignL -> AssignIsCExpr e CExprL
3
4 instance (AssignIsCExpr :<: f) => InjF (Term f) AssignL CExprL where
5   injF = AssignIsCExpr
6   projF x = case project x of
7     Just (AssignIsCExpr x) -> Just x
8     _                      -> Nothing
```

Figure 3-3: An example sort injection node and its associated sort injection

A sort injection node of sort `s` which contains a single child of sort `t` induces a sort injection from `t` to `s`. Any signature which contains this sort injection node as a summand then allows the definition of functions injecting and projecting between terms of sort `s` and `t`, which may be provided through a Haskell typeclass. An example sort injection node, and the typeclass instance providing its associated sort injection, are given in Figure 3-3. Sort injections may also be provided through a chain of existing nodes rather than through a dedicated sort injection node. The insertion of assignments into blocks described in Section 2.2 is an excellent example of such a "derived" sort injection. A transformation may then be written against any language which provides a sort injection from assignments to block items, abstracting over multiple syntaxes.

The inclusion of sort injection nodes in a representation allows it to modularly configure the interaction of components. For instance, our generic component for assignments has children of sorts `Lhs`, `Rhs`, and `AssignOp`, and is itself of sort `Assign`. An incremental parametric syntax which wishes to include the assignment component will then include sort injection nodes from its own relevant language-specific sorts into `Lhs`, `Rhs`, and `AssignOp`, and from `Assign` to wherever assignments may be inseretd in the language. For example, C and Java contain sort injection nodes from `Assign` to `CExpression` and `JavaExpression`, while Lua contains one from `Assign` to `LuaStatement`.

Chapter 4

Implementation

Our system is organized as a collection of libraries which users can use to assist in building incremental parametric syntaxes and multi-language transformations. The architecture of our system is given in Figure 4-1.

Our implementation totals approximately 10,000 lines of Haskell, providing support for five languages and several transformations. We build heavily on the `compdata` library [2] for representing modular syntax, and extend it with support for our new concept of sort injections and the `comptrans` library for converting a third-party syntax definition into modular syntax. We provide a small library of generic language components, and modules to assist in dealing with labeled terms, control-flow graphs, and higher-order tree traversals.

The code is split between approximately 3500 lines in our language implementations, 1200 in our transformations and their harness, 1000 in `comptrans`, 3300 in our other libraries, and the rest in various examples, tests, 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.

4.1 Languages

In addition to our core libraries, we implement support for C, Java, JavaScript, Lua, and Python. These use the parsers, pretty-printers, and syntax definitions from the

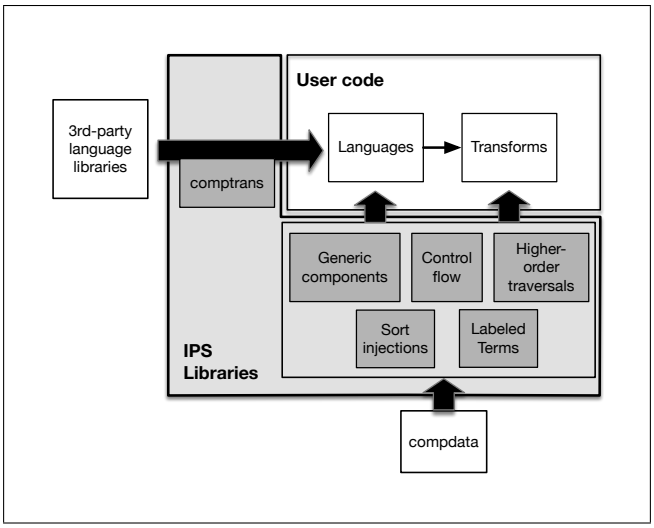


Figure 4-1: The architecture of the IPS libraries.

Haskell libraries `language-c` [17], `language-java` [5], `language-javascript` [43], `language-lua` [31], and `language-python` [32], respectively. Despite their names, these libraries were all implemented independently by different authors, and share no common infrastructure beyond the Haskell standard libraries. We fixed bugs in all of their pretty printers, but otherwise had no involvement with their development. Some of our fixes have yet to be merged upstream.

One exception is that, in a previous project, we had already built a translation between the JAPA parser [37] and the `language-java` AST, as the JAPA parser is substantially higher quality that `language-java`'s. We used this instead of the `language-java` parser. However, our translation is built on a version of JAPA for Java 1.5. Because some samples in the RWUS suite (Section 5.2.1) use Java 1.8 syntax, our implementation falls back to the `language-java` parser if JAPA fails.

Chapter 5

Evaluation

In this section, we examine the ability of our system to effectively build high-quality multi-language transformations. We make the following four claims:

- *Ease*: Transformations in our system are easy to write.
- *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
- *Completeness*: Despite the low effort needed per language, we can still build transformations that fully cover the language constructs, and perform the transformation as intended

We cover each of these aspects in their own section. In Section 5.1, we give a detailed case study of writing a transformation and evolving a transformation from one language to five. In Section 5.2, we present our human study, which gives evidence that the output of these transformations is more readable than hand-transformed code. Section 5.3 shows that these transformations preserve semantics on exhaustive suites of compiler tests, with relatively small improvements needed to bring success

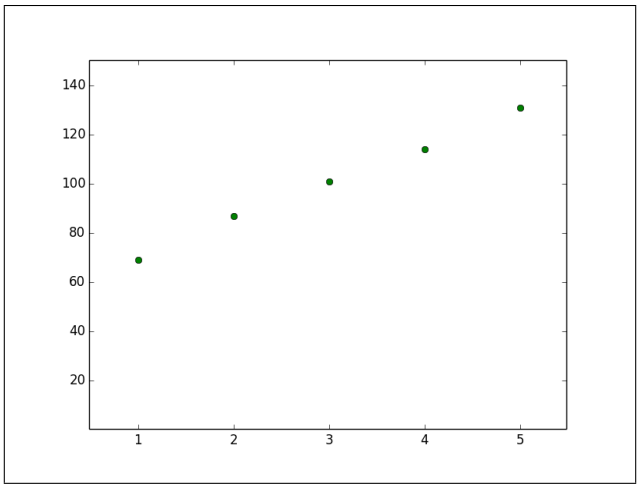


Figure 5-1: Lines of code needed as we extended the test-coverage transformation to Lua, C, Java, Python, and JavaScript, in that order.

rates to 100%. Finally, in Section 5.4 we report on our evaluations from manually inspecting transformed programs.

5.1 Ease: Case Studies

We described our first example transformation, declaration hoisting, in Section 2.1. In the following section, we present the test coverage instrumentation transformation, and describe how we were able to grow it from working on one language to five with almost no modification.

5.1.1 Case Study: Test Coverage Instrumentation

Our second example transformation is the test coverage instrumentation transformation. The goal of the test coverage transformation is to maintain a global array with one entry per basic block of the program, such that one can infer from that array which lines of code have executed. An example is given in Figure 5-2. The inspiration for this is the test coverage tool produced by the company Semantic Designs, which contains this transformation implemented separately for over a dozen languages [34].

The main idea of the transformation is simple: at the start of each basic block, in-

sert a line of code to mark the relevant entry in that array. Behind this lies substantial control-flow machinery.

Our implementation simply iterates through each node in a program, checks the control-flow graph to see if it starts a basic block, and, if so, inserts a statement to mark the global array.

We built a control-flow library which contains cookie-cutter modules to build CFG fragments for common cases. The library also allows a language to specify a list of sorts, and will generate code implementing default CFG construction fragments for those cases. Using this, we built a control-flow graph generator for each language. For each language, we also implemented an `InsertBefore` typeclass, which abstracts over the ability to insert term before a given node.

The transformation includes a constraint that each language must implement these. The transformation has a constraint that each language must implement the `MarkBlockCovered` typeclass, which provides an operation to generate an assignment into the global array.

The test coverage transformation gives an excellent case study in the ease of adding more languages to a transformation. We initially built the control-flow library and test-coverage transformation with Lua in mind. In less than two days, we were able to expand it to work on all five languages. The control-flow library increased in size by less than 100 lines of code as we added another four control-flow graph generators. Meanwhile, we found that we needed to change the core transformation exactly once, in order to account for languages with 0-indexed versus 1-indexed arrays. We handled this by creating a `BlockCounterStart` typeclass. The typeclass and instance declarations total 9 lines, as all languages but Lua use the default of 0. We added 1 line to the constraints of the transformation, and changed one line in the code itself.

Most of the work in the test-coverage transformation lies in the control-flow and insertion infrastructure. Our core test coverage transformation totals 47 lines, plus an average of 15 lines per language for the `MarkBlockCovered` and `BlockCounterStart` typeclass declarations. The control-flow graph builders average 80 lines of code per language, with substantial code cloning between them. The `InsertBefore` typeclass declarations

```
1 def countF():
2  count = 0
3  for i in
       range
       (100):
4   if f(i):
5    count += 1
6    break
7   else:
8    print i
9  return count
```

```
1 def countF():
2  TC.cov[0] =
       True
3  count = 0
4  for i in
       range
       (100):
5   TC. cov[1] =
        True
6   if f(i):
7    TC. cov[2]
         =
        True
8    count += 1
9    break
10  else:
11   TC. cov[3]
         =
        True
12   print i
13 TC. cov[4] =
       True
14 return count
```

Figure 5-2: Example of the test coverage instrumentation transformation for Python

and helper functions average 20 lines of code per language. The control-flow graph library itself is approximately 600 lines of code. These counts all omit the Haskell prologue of each file, which consists entirely of pragmas, import statements, and export declarations.

As always, there is a tradeoff between language-specific code and a unified representation. By modifying our incremental parametric syntaxes to use generic notions of loops and if-statements, we could substantially reduce the size of each control-flow generator.

5.2 Readability: Human Study

We ran a study to evaluate the readability of our transformations' output. The overall setup of our experiment is like a Turing test. First, we ask a set of human contributors to transform programs by hand. We then give a separate set of human judges these programs, alongside the corresponding automatically transformed programs, and ask them to rate them both on correctness and quality. Because low-level code formatting is outside the scope of our claims, we automatically reformat the human-written code before presenting them for comparison. Outside of formatting, we attempted to bias the experiment in favor of the humans, allowing them to resubmit until their transformed programs were correct according to our extremely thorough test suites. Despite this, in our final results, the judges gave the automatically transformed programs a higher average rating for all five languages.

Our experiment proceeds in three phases. In the first phase, we construct the RWUS suite, providing suitable programs on which to run the study. In the second phase, we ask human participants to manually apply each of the two studied transformations on a code sample. In the final phase, human judges from Mechanical Turk rate the manually-transformed code against the same code transformed by our system.

5.2.1 Phase 1: Constructing 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 semantically equivalent to the original. The second criterion is the main difficulty, as random samples of code typically do not come with thorough tests, and certainly not tests that are easy to run. Hence, we created our own.

The RWUS (Real World, Unchanged Semantics) suite consists of 50 functions across 5 languages randomly selected from top GitHub projects. For each, it also includes a test suite designed with the intention that only functions semantically equivalent to that function will pass. Each function is distributed as an *entry*. An

entry is a file containing the original sample, mocks for all referenced symbols, tests, and a wrapper `main` procedure which invokes the tests. The files can all be compiled and executed without any dependencies. The tests are used by invoking a script that replaces the sample with a transformed version, and then executes the resulting file.

We selected the functions for the RWUS suite as follows: For each of C, Java, JavaScript, Lua, and Python, we downloaded the top 20 projects in that language on Github from those with at least 500 lines, sorted by number of users who "starred" that project. We then uniformly at random selected a line of code from the projects. If this line of code lies within a function, we took the innermost such function as a sample; else, we repeated the process. We discarded all samples which were not between 5 and 50 lines of code, excluding function signatures, blank lines, and comments. We repeated this process until we had 10 samples for each language. One shortcoming of this approach is that the top-rated projects on Github vary in size by orders of magnitude. As the extreme, 90% of our C corpus and all 10 C samples come from Linux. The other 40 samples come from 24 different projects.

For each sample, we constructed test cases ensuring full path coverage, and added checks to ensure all mocked functions are called in the expected order with the expected arguments. The resulting tests are incredibly thorough. While the actual samples total 1158 lines of code, the RWUS suite totals 8070 lines of code.

5.2.2 Phase 2: Obtaining Human-Written Transformations

We recruited programmers through department mailing lists, flyers posted around the department, and social media. Due to the relative scarcity of Lua programmers, we also posted on Lua forums, and asked Lua participants to spread knowledge of the study by word of mouth.

Participants were sent to a website, where they would would download a single sample from the RWUS suite along with its tests, and were asked to perform each of our transformations by hand on the file. They were allowed to contribute one sample per language, and were offered a $10 Amazon gift card for each.

We inspected each submission by hand. Participants were asked to resubmit until

	C	Java	JS	Lua	Python
Identical	6	9	1	4	3
Failed	0	1	1	0	0

Table 5.1: Counts of programs where presentation to the human judges was inappropriate

their transformed samples passed all tests, and had no significant transformation errors, such as unhoisted variables.

5.2.3 Preparing the Samples

After we had collected all 50 human-transformed samples, we ran them through the corresponding parser and pretty printer to normalize formatting. We then ran our transformations on each of the RWUS samples, and evaluated them with the RWUS test suites.

We did not run any transformation on the RWUS samples until all development on the transformations had ceased. We also attempted to avoid allowing knowledge of the samples in the RWUS suite to influence development of the transformations, although a single researcher was responsible for both.

Of the 90 transformed pairs, for 24 of them, the automatically transformed version was identical to the human written one after reformatting. These are broken down per language in Table 5.1. Two automatically transformed samples failed their test suites, while, for one sample, a pretty-printer bug caused both the human-transformed and automatically transformed versions to fail to compile. The remaining 63 pairs were sent to human judges for evaluation in Phase 3.

5.2.4 Phase 3: Comparing Human and Machine-Written Transformations

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.

We created one task on Mechanical Turk for each language/transformation combination. For each judge entering our website interface, we began by presenting an explanation and example of the transformation, before bringing them to the questions. In each question, they were given a sample program, along with the automatically transformed version produced by our system, and the manually-transformed version collected in Phase 1, and were asked to rate both on a 1–5 scale. We instructed that they should first rate the transformed programs on correctness vs. the original program, second on faithfulness to the intended transformation, and only third on general prettiness and code quality. Both the order of questions and the order of the transformed pairs were randomized.

5.2.5 Quality Control

The setup described above does not preclude someone from rating programs randomly, so we employed two quality-control mechanisms. Our primary form of quality control was the creation of "canary" questions. Canaries appear as normal questions, except that the programs contained therein were contrived. In two of the canaries, one of the programs was clearly not a transformed version of the original. In the third canary, both displayed programs were identical. We rejected any submission in which the worker did not rate the correct program higher for the first two canaries, or did not rate both programs of the third canary the same. Second, if a worker ever submitted two answers within 11 seconds of each other, we marked this worker as untrustworthy, and rejected all submissions by him. We picked this value after observing the times spent on each question in dry runs of the study.

We noticed substantial differences between workers who did and did not pass the quality controls. Workers with one rejected submission typically had rejected submissions for many different languages. Workers with accepted submissions were much more likely to only submit for one language. Workers typically either had all their submissions accepted or all rejected. Furthermore, we noticed that rejected submissions were typically completed in much less time than accepted ones, although many workers who failed the canaries were substantially slower than the fastest correct

	C	Java	JS	Lua	Python
4	2	1	5	0	1
3	4	3	9	0	2
2	9	14	19	8	15
1	21	27	14	18	16
0	23	6	13	12	16
-1	24	8	11	10	25
-2	11	2	13	3	7
-3	1	5	2	3	0
-4	1	3	0	0	2

Table 5.2: Counts of differences between the ratings of the machine transformations and the human transformations. The higher rows represent cases where the judge rated the machine-produced output higher than the human-produced.

workers.

The experimenters manually inspected a selection of judgments from accepted submissions, and found them all reasonable. Overall, our observations suggest that our quality control mechanisms did effectively classify workers on skill, and that our data is high-quality.

5.2.6 Results

For each language, we tabulated the difference in ratings between the human-written and automatically transformed programs. Our results are given in Table 5.2. For all 5 languages, the humans rated the automatically transformed program higher on average. Combined with our results showing that the human and machine transformed programs were identical in 24/90 pairs, this presents strong evidence that our multi-language transformations built with our system produce human-quality output.

5.2.7 Threats to Validity

Our results are potentially biased by using a real-world distribution of programming constructs, as opposed to intentionally constructing a suite filled with corner cases. The humans are hindered by a lack of learning: they only perform each transformation once per language. Finally, we cannot be certain of the quality of the data from

Mechanical Turk. In our dry runs, we found that workers on Mechanical Turk tend to rate simple programs more highly, even when the transformation is incorrect. Two of our canaries are specifically designed to prevent this behavior.

5.3 Correctness

As the third part of our evaluation, we claim that incremental parametric syntax makes it possible to write transformations which are semantics-preserving, even against the complexities of multiple languages. We ran our transformations on language test suites for each of the 5 languages, and checked whether they still passed. Our results are promising, showing a 99.8% pass rate for the hoisting transformation, and 99.1% for the test coverage.

A caveat is that we depend on 3rd-party parsers and pretty printers, which exhibit incorrect behavior on many of these tests. We hence could only evaluate our transformations on the tests for which they worked correctly. Additionally, many of the tests are self-referential, and can fail if the formatting changes. Although we fixed some bugs in all of our pretty-printers, only 90% of tests passed the identity transformation, consisting of a simple parse+pretty-print pass.

Table 5.3 lists the language implementations and test suites used in our evaluation. The C, Java, and Lua tests come from their respective implementations, while the JavaScript ones come from the official specification conformance test suite. The authors of K-Java report that no Java language tests are publicly available [4], and hence created their own specification tests, which we use. We restricted ourselves to the core language tests of `test262`, 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.

The C, Java, and JavaScript tests are each distributed as standalone files, effectively containing a single test. The Python ones are built using PyTest. Each test file contains potentially dozens of tests, but we report at file granularity: if any tests in that file fail, we report that the file fails. The Lua ones are described below.

Table 5.4 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 test cases which fail due to bugs in our underlying parsers and pretty-printers, or due to overly self-referential test-cases. The **Hoist** and **Testcov** columns show the results of our hoisting and test-coverage instrumentation transformations, described in sections 2.1 and 2.1. The **TAC**, or **three-address code** transformation, is a transformation that lifts all nested computations into temporary variables, so that at most one computation is done per line. This is a complicated transformation, difficult to build even for one language, which must handle subtleties such as short-circuiting operators and hoisting loop conditions to multiple places Our implementation only works with JavaScript, Lua, and Python, because declaring the temporary variables in C and Java requires type-inference, which in turn requires more heavyweight infrastructure (e.g.: symbol-table construction).

While only 93.8% of the language tests work with our 3rd-party infrastructure, of those, our hoist transformation passes 98.2%, our test coverage passes 97.9%, and our TAC transformation passes 95.6%. Our inspection shows the failures result from minor bugs that would require small changes to correct, similar to the ones mentioned in Section 2.1. Furthermore, one of the failing Java hoist tests resulted in a crash of `javac`. Manual inspection shows that this program is indeed correct. The bug has been confirmed by the JDK developers [18].

We had substantial issues working with the Lua test suite. The Lua test suite is distributed as a single test program, containing 28 files and 12,000 lines of code. The tests are highly self-referential, and include checks that a symbol is defined on a certain line number, as well as checks that the current file has a certain character in a certain position. The test suite also makes heavy use of the dynamism of Lua, including multiple points where it undefines every global variable. Furthermore, the test suite is designed to stop at the first failed assertion. With tests that break if a file changes character encoding, let alone a format-altering identity transformation, using them to test correctness of our transformations would be a substantial endeavor.

Lang	Implementation	Test Suite
C	GCC 6.3.0_1	gcc-torture [16]
Java	JDK 1.8.0_65	K-Java [4]
JS	Node.js v0.10.24	test262 [36]
Lua	Lua.org 5.3.3	Lua Tests [33]
Python	CPython 3.7.0a0	CPython Tests [14]

Table 5.3: Language implementations and test suites used in our evaluation

Lang	Total	Ident	Hoist	Testcov	TAC
C	1394	1304	1228	1280	N/A
Java	755	738	*732	726	N/A
JS	2638	2478	2478	2414	2420
Python	404	353	N/A	345	293
Lua	Reported separately				
* Not including test which crashed `javac`					

Table 5.4: Results running each transformation on the language test suites

We decided to modify the Lua test suite to maintain a count of passed assertions, instead of stopping at the first failure, and to delete some of the overly self-referential assertions. In our resulting modified test suite, we found that the total number of calls to `assert` was nondeterministic, 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 easily overwhelms this on the test suite. We conclude that the Lua test suite is overall unsuitable for testing program transformations.

5.4 Completeness

With no objective criteria to evaluate whether our transformations transform the program as intended, we resorted to researcher judgment. For every language and

transformation, we manually inspected 10 of the transformed programs from the relevant language test suite. We also inspected a sample of all failing test cases from Section 5.3. Our overall judgment was that the programs correctly transformed an impressive variety of programs. We found 12 bugs in the transformation, including both places where it failed to transform the program, or inserted a line of code illegally. The largest of these is that we have only partially implemented a subtlety in the hoisting transformation for JavaScript, in that declarations should be hoisted to the top of the containing function, because inner blocks do not delimit scopes. For each bug, we estimated what level of effort would be required to fix it. We estimated that each of these would require relatively small changes involving little restructuring, thus giving us increased confidence that our transformations can handle the subtleties of many languages, without paying the price of an implementation per language.

Chapter 6

Related Work

Our work is most directly based on the data types à la carte approach to modular syntax [35], and its extensions in Bahr's work on compositional datatypes [2]. The extension of data types à la carte to multi-sorted terms was introduced in Yakushev et al [41]. Other approaches to modular syntax include tagless-final [20], object algebras, and modular reifiable matching [30].

This work in modular syntax is joined by work in modular semantics, such as modular monadic semantics [24] and its proof-theoretic incarnation modular monadic meta-theory [10], as well as modular structural operation semantics [28].

Sort injections are an instance of the concept of *feature interactions* from the field of software product lines [38].

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 [19], Rascal [21], TXL [8], Semantic Designs DMS [3], and JetBrains MPS [39]. These were extensively surveyed in Erdweg et al [13]. 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.

One recent work that echoes our own is Brown et al's [6] work using *island grammars* [27] 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 [12] is a technique for using island grammars to construct parsers. Their work focuses on concrete syntax (i.e.: parsing), while ours focuses on abstract syntax (i.e.: representation).

Bibliography

[1] Robert L. Akers, Ira D. Baxter, Michael Mehlich, Brian J. Ellis, and Kenn R. Luecke. Case study: Re-engineering C++ component models via automatic program transformation. *Information & Software Technology*, 49:275–291, 2005.

[2] Patrick Bahr and Tom Hvitved. Compositional data types. In *ICFP*, 2011.

[3] Ira D Baxter, Christopher Pidgeon, and Michael Mehlich. DMS®: Program transformations for practical scalable software evolution. In *Proceedings of the 26th International Conference on Software Engineering*, pages 625–634. IEEE Computer Society, 2004.

[4] Denis Bogdanas and Grigore Roşu. K-Java: a complete semantics of Java. In *ACM SIGPLAN Notices*, volume 50, pages 445–456. ACM, 2015.

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

[6] Fraser Brown, Andres Nötzli, and Dawson Engler. 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*, pages 143–157. ACM, 2016.

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

[8] James R Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

[9] Benjamin Delaware, Steven Keuchel, Tom Schrijvers, and Bruno C. d. S. Oliveira. Modular monadic meta-theory. In *ICFP*, 2013.

[10] Benjamin Delaware, Steven Keuchel, Tom Schrijvers, and Bruno CdS Oliveira. Modular monadic meta-theory. *ACM SIGPLAN Notices*, 48(9):319–330, 2013.

[11] Isil Dillig, Thomas Dillig, and Alex Aiken. SAIL: Static Analysis Intermediate Language with a Two-level Representation. 2009.

[12] Tom Dinkelaker, Michael Eichberg, and Mira Mezini. Incremental concrete syntax for embedded languages with support for separate compilation. *Science of Computer Programming*, 78(6):615–632, 2013.

[13] 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. The state of the art in language workbenches. In *International Conference on Software Language Engineering*, pages 197–217. Springer, 2013.

[14] Python Software Foundation. CPython Test Suite. Version 3.7.0a0. `https://docs.python.org/devguide/runtests.html`.

[15] Martin Fowler. Refactoring: Improving the Design of Existing Code. In *XPU*, 1999.

[16] FSF. C language testsuites: "C-torture". Revision 240758. `http://gcc.gnu.org/onlinedocs/gccint/C-Tests.html`, October 2016.

[17] Benedikt Huber. language-c: Analysis and generation of C code. `http://hackage.haskell.org/package/language-c`.

[18] JDK Bug System. javac crash when local from enclosing context is captured multiple times. `https://bugs.openjdk.java.net/browse/JDK-8169345`.

[19] Lennart CL Kats and Eelco Visser. *The spoofax language workbench: rules for declarative specification of languages and IDEs*, volume 45. ACM, 2010.

[20] Oleg Kiselyov. Typed tagless final interpreters. In *Generic and Indexed Programming*, pages 130–174. Springer, 2012.

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

[22] Ralf Lammel and Chris Verhoef. Cracking the 500-language problem. *IEEE software*, 18(6):78–88, 2001.

[23] Chris Lattner and Vikram S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, 2004.

[24] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343. ACM, 1995.

[25] Sheng Liang, Paul Hudak, and Mark P. Jones. Monad Transformers and Modular Interpreters. In *POPL*, 1995.

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

[27] Leon Moonen. Generating robust parsers using island grammars. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 13–22. IEEE, 2001.

[28] Peter D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.

[29] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC*, 2002.

[30] Bruno C d S Oliveira, Shin-Cheng Mu, and Shu-Hung You. Modular reifiable matching: a list-of-functors approach to two-level types. *ACM SIGPLAN Notices*, 50(12):82–93, 2016.

[31] Ömer Sinan AÄ§acan and Eric Mertens. language-lua: Lua parser and pretty-printer. `http://hackage.haskell.org/package/language-lua-0.10.0`, August 2016.

[32] Bernard James Pope. language-python: Parsing and pretty printing of Python code. `http://hackage.haskell.org/package/language-python-0.5.4`, July 2016.

[33] PUC-Rio. Lua: Test suites. Version 5.3.3. `https://www.lua.org/tests/`.

[34] Semantic Designs, Inc. Test Coverage tools. `http://www.semanticdesigns.com/Products/TestCoverage/`.

[35] Wouter Swierstra. Data types à la carte. *Journal of functional programming*, 18(04):423–436, 2008.

[36] Ecma TC39. Test262: Ecmascript language conformance test suite. version 5.1. `http://test262.ecmascript.org`, 2014.

[37] Danny van Bruggen. JavaParser: Process Java code programmatically. `http://javaparser.org`.

[38] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 45–54. IEEE, 2001.

[39] Markus Voelter and Vaclav Pech. Language modularity with the MPS language workbench. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1449–1450. IEEE, 2012.

[40] Philip Wadler. The expression problem. *Java-genericity mailing list*, 1998.

[41] Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *ACM Sigplan Notices*, volume 44, pages 233–244. ACM, 2009.

[42] Haoyuan Zhang, Zewei Chu, Bruno C. d. S. Oliveira, and Tijs van der Storm. Scrap your boilerplate with object algebras. In *OOPSLA*, 2015.

[43] Alan Zimmerman. language-javascript: Parser for JavaScript. `http://hackage.haskell.org/package/language-javascript-0.6.0.9`, November 2016.