A Language Designer's Workbench

A One-Stop-Shop for Implementation and Verification of Language Designs

Eelco Visser
Delft University of Technology
visser@acm.org

Guido Wachsmuth
Delft University of Technology
guwac@acm.org

Andrew Tolmach
Portland State University
apt@cs.pdx.edu

Pierre Neron, Vlad Vergu
Delft University of Technology
{p.j.m.neron, v.a.vergu}@tudelft.nl

Augusto Passalaqua, Gabriël Konat
Delft University of Technology
{a.passalaquamartins, g.d.p.konat}@tudelft.nl

Abstract

The realization of a language design requires multiple artifacts that redundantly encode the same information. This entails significant effort for language implementors, and often results in late detection of errors in language definitions. In this paper we present a proof-of-concept language designer's workbench that supports generation of IDEs, interpreters, and verification infrastructure from a single source. This constitutes a first milestone on the way to a system that fully automates language implementation and verification.

Categories and Subject Descriptors D.2.6 [*Software Engineering*]: Programming Environments; D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.2 [*Programming Languages*]: Language classifications; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.3.4 [*Programming Languages*]: Processors; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages

Keywords Language Designer Workbench; Meta-Theory; Language Specification; Syntax; Name Binding; Types; Semantics; Domain Specific Languages

1. Introduction

Programming language design is hard. But we continue to need new languages, particularly domain-specific languages (DSLs), to help software engineers handle the complexity of modern software systems. Programming language designers want only one thing: to get usable, reliable realizations of their languages into the hands of programmers as efficiently as possible. To achieve this goal, they need to produce a number of artifacts:

- A compiler or interpreter that allows programmers to execute programs in the language;
- An IDE that supports programmers in constructing programs in the language;
- A high-level specification of the language that documents its intent for programmers;
- Validation, via automated testing or formal verification, that their language designs and implementations are correct and consistent.

Today's savvy language designer knows that there are good tools available to help with these tasks. However, existing tools generally require the designer to create each of these artifacts separately, even though they all reflect the same underlying design. Consequently, a compiler or interpreter is often the only artifact produced; documentation, IDE, and—especially—validation are typically omitted. For example, language implementations seldom formally guarantee semantic correctness properties such as type soundness and behavior preservation of transformations, because current implementation tools provide no support for verification. This can lead to subtle errors in languages that are discovered late.

Some state-of-the-art tools do support the production of two or more language artifacts. Language implementation workbenches such as Xtext [54], MPS [24], and Spoofax [26] provide fairly high-level support for implementation of IDEs and code generators. Semantics engineering tools such as K [15], Redex [17], and Ott [46] support the high-level specification of type systems and dynamic semantics in order to facilitate testing or verification. The CompCert

Onward! 2014, October 20–24, 2014, Portland, OR, USA.

ACM 978-1-4503-3210-1/14/10.
http://dx.doi.org/10.1145/2661136.2661149

certified C compiler [32] is both implemented and verified within the Coq theorem prover. But no current tool addresses the full scope of designer activities.

Our vision is a language *designer's* workbench as a one-stop-shop for development, implementation, and validation of language designs. The key idea for the realization of this vision is to conceptualize the sub-domains of language definition as a collection of declarative language definition formalisms (or *meta-languages*) that are multi-purpose, so that a single language definition can be used as the source for the implementation of efficient and scalable compilers and IDEs, the verification or testing of correctness properties, and as a source of (technical) documentation for users of the language.

In this paper, we report on a first milestone towards the goal of supporting the complete specification of syntax, static semantics, and dynamic semantics within a single integrated environment. To this end, we have extended the Spoofax Language Workbench[1], which already supported declarative specification of syntax definition and name binding, with declarative meta-languages for type analysis and dynamic semantics (Section 4). From specifications in these meta-languages we derive (i) an IDE including syntactic editor services such as syntax checking, syntax highlighting and semantic editor services such as type checking and reference resolution (Section 5), (ii) an interpreter (Section 6), and (iii) Coq definitions that encode semantics in a form that can be used to verify language properties (Section 7). (We leave support for automated testing to future work.)

The contributions of this paper are:

- The architecture of a language designer's workbench that separates the concerns of language definition into syntax definition, name binding, type analysis, and dynamic semantics specification (Section 3).
- The TS language for type analysis (Section 4.3), which complements the specification of name binding and scope rules in our existing NaBL language.
- The DynSem language for the specification of operational semantics (Section 4.4), based on the work on modular operational semantics by Mosses [36, 9].
- The systematic derivation of a naive interpreter in Java from a dynamic semantics specification in DynSem (Section 6).
- The systematic derivation of Coq definitions for syntax definition, name binding rules, type analysis rules, and dynamic semantics rules (Section 7). In particular, we provide a semantics that characterizes well-bound terms according to a specification in a subset of NaBL.
- The specification of the syntax and semantics of the PCF language in these meta-languages (Section 4) and the generation of an IDE, interpreter, and proof infrastructure from that specification. Based on the latter we manually prove type preservation for the dynamic semantics (Section 7).

The version of the workbench that we present here is a *proof of concept*. While we expect that the approach can be applied to a wide range of languages, we have not yet validated that expectation. It is likely that our meta-languages, in particular the more experimental ones for type analysis and dynamic semantics, will be refined as we extend their coverage.

2. Problem

Bridging the gap between domain concepts and the implementation of these concepts in a programming language is one of the core challenges of software engineering. Modern programming languages have considerably reduced this gap, but still require low-level programmatic encodings of domain concepts. Domain-specific software languages (DSLs) address the complexity problem through *linguistic abstraction* by providing notation, analysis, verification, and optimization that are specialized to an application domain and allow developers to directly express design intent ('language shapes thought').

Rapidly developing DSLs for emerging domains of computation can become an important method to manage software complexity. However, a software language is a complex software system in its own right — consisting of syntactic and semantic analyzers, a translator or interpreter, and an interactive development environment (IDE) — and can take significant effort to design and implement. *Language Workbenches* are language development tools [18] that considerably lower the threshold for software engineers to develop DSLs to automate software tasks. Examples of modern language workbenches include MPS [24], Xtext [54], and Spoofax [26].

2.1 Correctness of Language Definitions

Language definitions and implementations rarely guarantee semantic consistency properties such as type soundness, correct capture-free substitution, or semantics preservation by transformations.

As the languages developed with workbenches grow in complexity, more subtle semantic errors will be introduced. These may result in type checkers that fail to signal programs that might cause run-time errors; code generators that produce ill-formed code; optimizers that produce code with a different meaning than the (intended) meaning of the source program; and refactorings that change the meaning of programs. For example, refactoring implementations often contain many bugs, even in mature tools such as Java IDEs [45]. The correctness of these tools depends on informal programmer reasoning, which easily overlooks corner cases and feature interactions. Errors that emerge may be hard to locate in the implementation, and their absence may be hard to guar-

[1] Spoofax Eclipse is available at `http://metaborg.org/wiki/spoofax`

antee. These problems are aggravated when languages are invented by software engineers who lack formal training in language design.

Formal verification can ensure that semantic correctness properties hold for all possible programs, which is necessary for complex transformations such as refactorings [45]. Property-based testing can aid in detecting errors by randomly generating programs to test semantic correctness properties [28, 55]. However, the focus of the current generation of language workbenches is on using language definitions to construct implementations; they provide no support for formal verification.

This situation is problematic at several levels. First, it affects language designers who spend time chasing bugs. Second, the late detection of problems affects the quality of language designs; repairing languages becomes hard once a body of code has been developed. Third, it affects the reliability of programming tools, which in turn affects the productivity of software developers and the quality of the software they produce. Overall, the resulting complexity of the language design and implementation process is an impediment to the large scale introduction of linguistic abstraction as a software engineering tool.

2.2 Analysis

Understanding this state-of-the-art requires a closer look at the nature of language design and implementation. A language implementation includes the following *language components*. A *compiler* translates a source program to executable code in a lower-level language. An *interpreter* or *execution engine* directly executes a program. An *integrated development environment (IDE)* supports developing (editing) programs by means of a range of editor services that check programs and help in navigating large programs.

The production of each of these components incorporates several or all of the following *language definition concerns*. A *syntax definition* defines the structure of well-formed sentences in the language. *Name binding and scope rules* determine the relation between definitions and uses of names in programs. A *type system* defines static constraints on syntactically well-formed programs, avoiding a large class of run-time errors. *Transformations* define modifications that improve programs in some dimension such as performance or understandability. The *dynamic semantics* defines the dynamic behaviour (execution) of programs. A *code generator* defines the translation to code in another, typically lower-level, 'target' language.

Over five decades, programming language developers and designers have produced many different methods for encoding the definition of language concerns. Some are specialized to the implementation of compilers and IDEs, others are specialized to reasoning about the correctness of language definitions. This leads to redundancy and duplication of effort for language designers.

Specialization in language workbenches. For most of these concerns, existing language workbenches require designers to encode their intent by writing explicit code, and to do so separately for each concern. Syntax definitions are the exception to this rule [27]: they are declarative and can be used to address multiple concerns—characteristics that we would like to emulate in other domains if possible.

Programmatic encoding. Language definitions are encoded in a general-purpose programming language, which makes domain-specific analysis and verification difficult. For example, the implementation of name resolution in a programming language, or even as attribute grammar, obscures the definition of name binding concepts such as 'definition', 'reference', and 'scope'. This makes it impossible to check a property such as 'reachability of definitions from references'. By contrast, a context-free grammar is a declarative definition of the syntax of a language, which abstracts from the details of parser implementations, and which can be analyzed for properties such as reachability of non-terminals.

Specialized definitions. Definitions are specialized to specific components, which requires re-definition of the same concepts in different formats, increasing the risk of inconsistencies. For example, the name binding rules for a language are encoded in a name resolution algorithm for editor services such as code completion, and in a capture-free substitution function for use in transformations. By contrast, a declarative syntax definition can be used as the single source for deriving a parser, a pretty-printer, a language-aware editor, and an abstract syntax tree type definition.

Specialization in semantics engineering. Another line of tools for language designers aims at support for *semantics engineering*. Recent advances in executable definitions of dynamic semantics [17, 44], DSLs for semantics specification [46], model checking of type systems [43], random test generation for compiler correctness [11, 28, 55], and verified compilers [32], promise a *semantics engineering* discipline with automated testing and verification of language safety properties, detecting bugs early, and increasing the confidence in language definitions.

However, in their current form these approaches are not yet suitable for the assistance of software engineers designing DSLs for one or more of the following reasons: (i) Semantic definitions are aimed at *modeling* languages for the purpose of verification and ignore implementation constraints. For example, reference resolving in an IDE poses different constraints on the definition of name binding than the implementation of a type soundness proof. The result is redundancy in specification efforts. (ii) Realization of verification requires significant overhead with respect to the basic definitions. For instance, the verification code for the CompCert compiler [32] is about seven times as large as the compiler implementation itself. This overhead is aggravated

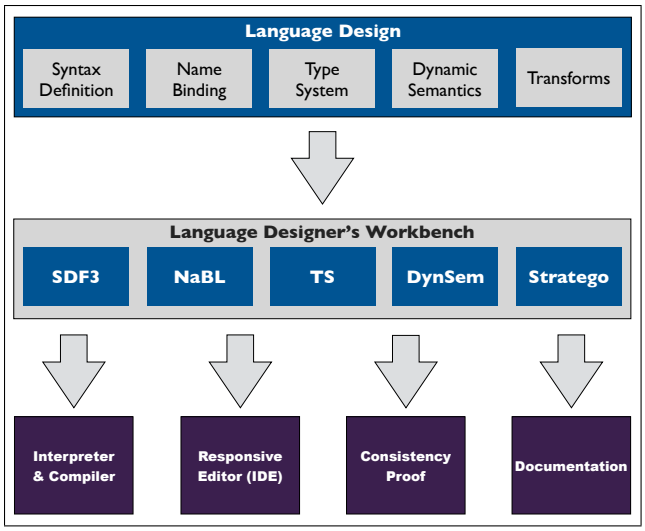

Figure 1. Architecture of a language designer's workbench. Language design is directly expressed using high-level declarative meta-languages from which multiple artifacts are derived automatically.

by use of low-level languages ('internal DSLs') for verification. "Proof assistants help with automatic checking, but come with their own problems. The sources of definitions are still cluttered with syntactic noise, non-trivial encodings are often needed [46]." (iii) Semantic definitions need to be reformulated for use with different proof assistants. This breaks the abstraction of declarative language definitions and exposes DSL designers to the 'implementation details of verification', which makes adoption by a larger audience of software engineers problematic. (iv) The approach is specific for a particular language or compiler and is not reusable for other languages. (v) The approach assumes low-level target languages, and does not address the complexity of domain-specific target platforms.

In our work we aim to adapt and extend the achievements of semantics frameworks and mechanized meta-theory, making them accessible to software engineers using language workbenches to design DSLs.

3. Architecture

A *language designer's workbench* is a programming environment that supports language designers in all aspects of language design, eliminates redundancy, and automates implementation and verification. Our approach to the realization of such a workbench is illustrated by the architecture in Figure 1.

Separation of concerns. The first goal is represented by the top arrow in Figure 1. Language designers should be able to directly express their designs rather than programmatically encoding them. This requires conceptualizing the sub-domains of language definition as a collection of language definition formalisms: specialized meta-languages—

PCF.sdf3

```
templates

  Exp.Var = [[ID]]
  Exp.App = [[Exp] [Exp]] {left}
  Exp.Fun = [
    fun [Param] (
      [Exp]
    )
  ]
```

names.nab

```
binding rules

  Var(x) :
    refers to Variable x

  Param(x, t) :
    defines Variable x of type t

  Fun(p, e) :
    scopes Variable
```

types.ts

```
type rules // binding

  Var(x) : t
  where definition of x : t

  Param(x, t) : t

  Fun(p, e) : FunType(tp, te)
  where p : tp and e : te

  App(e1, e2) : tr
  where e1 : FunType(tf, tr) and e2 : ta
    and tf == ta
        else error "type mismatch" on e2
```

semantics.ds

```
rules // binding

  E env |- Var(x) --> v
  where env[x] = T(e, env'),
        E env' |- e --> v

  E env |- Fun(Param(x, t), e) --> C(x, e, env)

  E env |- App(e1, e2) --> v
  where E env |- e1 --> C(x, e, env'),
        E {x |--> T(e2, env), env'} |- e --> v
```

example1.pcf

```
let fac : int -> int =
  fix f : int -> int (
    fun n : int (
      ifz n then 1
      else n * f (n - 1)
    )
  )
in (fac 3)
```

Figure 2. Definition of PCF in Spoofax Language Workbench with, from top to bottom, syntax definition in SDF3 (Figure 4), name binding rules in NaBL (Figure 5), type rules in TS (Figure 6), dynamic semantic rules in DynSem (Figure 7), and generated editor for PCF (Figure 3).

for syntax definition, name binding, type analysis, dynamic semantics, and transformation—that abstract from specific implementation concerns.

Multi-purpose language definition. The second goal is represented by the bottom arrows in Figure 1. The expression of a design should be multi-purpose. A language definition should be usable as the single source for the implementation of efficient and scalable compilers and IDEs, the verification of correctness properties, and as a source of (technical) documentation for users of the language. We believe this goal is also reachable through separation of concerns; by specializing meta-languages we can remove any bias towards the implementation of a particular artifact.

Proof of concept. The Spoofax Language Workbench [26] provides a platform for the construction of IDEs based on declarative syntax definition with SDF2 [48] and program transformation with Stratego [5]. We are in the process of refactoring and extending Spoofax to realize the architecture sketched in Figure 1. For that purpose, we are developing the meta-languages SDF3 for syntax definition, NaBL for name binding, TS for type analysis, and DynSem for dynamic semantics to support the complete specification of syntax and semantics of languages.

Here we present a first proof of concept that exercises all aspects of the new approach for a very basic language. We have formalized the syntax, static semantics, and dynamic semantics of PCF [35] in the new meta-languages[2]. Figure 2 shows the definition of PCF edited in Eclipse/Spoofax together with the generated IDE for PCF. We discuss the formalizations together with the meta-languages in the next section. From this definition we generate an IDE (Section 5), an interpreter (Section 6), and definitions for the Coq proof assistant, which can be used as the basis for a proof of type preservation for the dynamic semantics (Section 7). We have not yet addressed compilation, code generation, and transformation in this new setting. Regarding documentation, our first concern is to design meta-languages that lead to readable language definitions. We have not yet addressed the automatic generation of documents that can serve as technical documentation of a language such as supported by Redex [17] and K [15].

4. Declarative Language Definition

In this section we describe the Spoofax meta-languages for syntax definition, name binding, type analysis, and dynamic semantics. As running example we use the PCF language [35]. Fig. 3 shows an example PCF program.

4.1 Syntax Definition

Often, the syntax of a language is only formally defined in terms of a parser that turns a program text into an abstract syntax tree. However, there are many other operations that depend on the grammar of a language, including pretty-printing, syntax highlighting, and syntactic completion. These are typically implemented as separate artifacts, duplicating information that is part of the grammar. Furthermore, parser-oriented approaches expose the underlying parsing technique. Declarative approaches to syntax definition allow reasoning about the structure of a language independently of the parsing technique enabled by generalized parsing algorithms [27]. In addition, such declarative syntax definitions can be interpreted for multiple purposes.

We are developing SDF3 as a successor to the syntax definition formalisms SDF [20] and SDF2 [48]. SDF3 fully integrates abstract syntax tree generation, derivation of pretty-printers, and syntactic completion templates [49]. Fig. 4 defines the syntax of PCF in SDF3.

Tree structure. An important principle guiding the design of SDF3 is to support understanding of the syntax of a language purely in terms of its tree structure [27]. An SDF3 production such as

```
Exp.Fun = [fun [Param] ([Exp])]
```

defines a context-free grammar production

```
Exp -> "fun" Param "(" Exp ")"
```

That is, it defines one alternative for the non-terminal (`Exp`) to be a sequence of symbols `[fun [Param] ([Exp])]` in which the anti-quoted identifiers (`[Param]` and `[Exp]`) correspond to phrases generated by the corresponding non-terminals. The constructor (`Fun`) is used to construct nodes in the abstract syntax tree corresponding to this production. Thus, the production implicitly defines a constructor with signature

```
Fun : Param * Exp -> Exp
```

Fig. 3 shows the term structure of the PCF program according to the grammar of Fig. 4.

Declarative disambiguation. Instead of encoding the disambiguation of expressions in the grammar rules by means of a non-terminal for each precedence level, SDF3 supports the declarative disambiguation of expressions using relative priorities[3]. For example, the declaration

```
Exp.Mul > {left: Exp.Add Exp.Sub}
```

states that multiplication has higher priority than addition and subtraction, which are mutually left associative. Furthermore, associativity annotations on productions indicate whether an operator is left, right, or non-associative.

Templates. Productions are called *templates* since grammars double as generators [49]. A template characterizes a syntax pattern, quoting the literal text in the pattern, and using anti-quotation to indicate the holes. The whitespace separating the symbols in the template body is interpreted as arbitrary `LAYOUT` for the purpose of parsing. It is interpreted as line break and indentation directives when producing syntac-

[2] Available at `https://github.com/metaborgcube/metaborg-pcf`

[3] For some corner cases additional non-terminals may still be needed.

```
let fac : int -> int =
  fix f : int -> int (
    fun n : int (
      ifz n then 1 else n * (f (n - 1))))
 in (fac 3)
```

```
Let("fac", FunType(IntType(), IntType()),
  Fix(
    Param("f", FunType(IntType(), IntType())),
    Fun(Param("n", IntType()),
      Ifz(Var("n"), Num("1"),
        Mul(Var("n"),
          App(Var("f"),
            Sub(Var("n"), Num("1")))))))),
  App(Var("fac"), Num("3"))
)
```

Figure 3. Factorial function in PCF concrete syntax and abstract syntax according to the syntax definition in Figure 4.

```
module PCF

imports Common

context-free start-symbols Exp

sorts Exp Param Type

templates

  Exp.Var = [[ID]]
  Exp.App = [[Exp] [Exp]] {left}
  Exp.Fun = [
    fun [Param] (
       [Exp]
    )
  ]
  Exp.Fix = [
    fix [Param] (
      [Exp]
    )
  ]
  Exp.Let = [
    let [ID] : [Type] =
       [Exp]
    in [Exp]
  ]
  Exp.Ifz = [
    ifz [Exp] then
      [Exp]
    else
      [Exp]
  ]
  Exp.Num = [[INT]]
  Exp.Add = [[Exp] + [Exp]] {left}
  Exp.Sub = [[Exp] - [Exp]] {left}
  Exp.Mul = [[Exp] * [Exp]] {left}
  Exp.Div = [[Exp] / [Exp]] {left}
  Exp     = [([Exp])] {bracket}

  Type.IntType = [int]
  Type.FunType = [[Type] -> [Type]]

  Param.Param = [[ID] : [Type]]

context-free priorities

  Exp.App > Exp.Mul > {left: Exp.Add Exp.Sub}
  > Exp.Ifz
```

Figure 4. Syntax definition for PCF in SDF3.

```
module names

namespaces Variable

binding rules

  Var(x) :
    refers to Variable x

  Param(x, t) :
    defines Variable x of type t

  Fun(p, e) :
    scopes Variable

  Fix(p, e) :
    scopes Variable

  Let(x, t, e1, e2) :
    defines Variable x of type t in e2
```

Figure 5. Name binding and scope rules for PCF in NaBL.

tic completion and pretty-print rules. Thus, pretty-printing schemes can be defined by example.

4.2 Name Binding

A key complication in any language engineering framework is the treatment of name binding. While much language processing can follow the inductive structure of abstract syntax trees, names cut across that structure with references to other parts of the tree. For example, a function definition introduces a name that can be used to call the function from a distant point in the tree.

In language engineering approaches, name bindings are often realized using a random access symbol table such that multiple analysis and transformation stages can reuse the results of a single name resolution pass [1]. Another approach is to represent the result of name resolution by means of *reference attributes*, direct pointers from the uses of a name to its definition [19]. Semantics engineering approaches to name binding vary from first-order representation with substitution [28], to explicit or implicit environment propagation [41, 36, 9], to higher-order abstract syntax [8]. Furthermore, some approaches use a nameless representation (De Bruijn index) of bound variables to uniquely represent terms modulo alpha equivalence [2]. In semantics engineering approaches, identifier bindings are represented with environments that are passed along in derivation rules, rediscovering bindings for each operation [41]. The key problem of all these approaches is the specification of name binding by means of an encoding in a more general purpose formalism (ranging from programming languages, to attribute grammars, to transition systems, to lambdas in the meta-language), which requires decoding for understanding and analysis, and is typically biased to a particular purpose (such as substitution, completion, navigation).

The references in a language are governed by rules for name binding and scope. The key concepts in these rules are

definitions that introduce names, *references* to definitions, and *scopes* that restrict the visibility of definitions. However, those rules are typically not directly expressed. Rather they are programmatically encoded and repeated in many parts of a language implementation, such as the definition of a substitution function, the implementation of name resolution for editor services, and refactorings. This repetition results in duplication of effort and risks inconsistencies.

We are developing *NaBL*, a high-level language for the declarative specification of the *name binding and scope rules* expressed directly in terms of name binding concepts [30]. Fig. 5 defines the name binding rules for PCF using the three basic NaBL concepts: *definitions*, *references*, and *scopes*. The rules associate name binding declarations with abstract syntax tree patterns. For example, the rule for the formal parameter of a function expression states that a `Param(x, t)` term is a definition for the variable `x`. Similarly, a `Var(x)` term is a *reference* to a definition of a variable with name `x`. Finally, the `Fun` and `Fix` rules state that these constructs are *scopes* for `Variable`s. This means that variables defined inside nodes constructed with `Fun` and `Fix` are only visible within the sub-terms dominated by those nodes. A more precise characterization of NaBL's rules is given in Section 7. The `Variable` qualifier that is used in the rules indicates the *namespace* of a name. While PCF has only one kind of name, i.e. variables, most programming languages distinguish multiple kinds (e.g. classes, fields, methods, variables in Java). Namespaces in NaBL keep these names distinct.

4.3 Type Analysis

Types "categorize objects according to their usage and behaviour [6]." A *type system* formalizes this categorization, in order to ensure that only the intended operations are used on the representation of values of a type, avoiding run-time errors [41]. Typical formalizations of type systems tangle the name resolution with type analysis, the computation of types of expressions. We are developing *TS*, a high-level language for the declarative specification of type analysis that is complementary to the name analysis expressed in NaBL. Fig. 6 defines the type rules for PCF in TS.

Type rules define judgments of the form `p : t` stating that term `p` has type `t`. Terms and types are abstract syntax trees based on the syntax definition. The `where` clause of a rule defines the conditions under which the judgment holds. For example, consider the rule for function application (`App`) in Fig. 6. The rule defines that `App(e1, e2)` has type `tr`, provided that the first argument `e1` has function type `FunType(tf, tr)`, the second argument `e2` has type `ta`, and that the formal argument and the actual argument have the same type (`tf == ta`). The `else` branch on that equation generates a type error for use in the IDE on target term `e2`.

Type rules are complementary to name binding. This means that type rules do not have to propagate typing environments. Assuming that name binding has been performed,

```
module types

type rules // binding

  Var(x) : t
  where definition of x : t

  Param(x, t) : t

  Fun(p, e) : FunType(tp, te)
  where p : tp and e : te

  App(e1, e2) : tr
  where e1 : FunType(tf, tr) and e2 : ta
    and tf == ta
        else error "type mismatch" on e2

  Fix(p, e) : tp
  where p : tp and e : te
    and tp == te
        else error "type mismatch" on p

  Let(x, tx, e1, e2) : t2
  where e2 : t2 and e1 : t1
    and t1 == tx
        else error "type mismatch" on e1

type rules // arithmetic

  Num(i) : IntType()

  Ifz(e1, e2, e3) : t2
  where e1 : IntType() and e2 : t2 and e3 : t3
    and t2 == t3
        else error "types not compatible" on e3

  e@Add(e1, e2)
  + e@Sub(e1, e2)
  + e@Mul(e1, e2) : IntType()
  where e1 : IntType()
        else error "Int type expected" on e
    and e2 : IntType()
        else error "Int type expected" on e
```

Figure 6. Type analysis rules for PCF in TS.

the type rule for variables (`Var`) can obtain the type of a variable by simply referring to the `definition of x`. Note that in the name binding rule for `Param` in Fig. 5 we associated the type from the parameter declaration to the definition of a variable.

The rules for PCF suggest that name resolution and type analysis can be staged sequentially. In general, however, name binding may depend on the results of type analysis. For example, in a class-based object-oriented language, resolving a method call expression `e.f(e1,..,en)` requires knowing the type of the receiver `e` to determine the target class, and may require knowing the types of the argument expressions to resolve overloading.

4.4 Dynamic Semantics

The *dynamic semantics* of a language describes the behavior of its programs. Current language workbenches do not provide dedicated support for specification of dynamic semantics. Efficient interpreters can be implemented in a reg-

```
module semantics

signature
  constructors
    C : string * Exp * E → value // closure
    I : int → value              // integer
    T : term * E → value         // thunk

    addInt : int * int → int { native }
    subInt : int * int → int { native }
    mulInt : int * int → int { native }

rules // binding

  E env ⊢ Var(x) → v
  where env[x] ⇒ T(e, env'),
        E env' ⊢ e → v

  E env ⊢ Fun(Param(x, t), e) → C(x, e, env)

  E env ⊢ App(e1, e2) → v
  where E env ⊢ e1 → C(x, e, env'),
        E {x ↦ T(e2, env), env'} ⊢ e → v

  E env ⊢ Fix(Param(x, t), e) → v
  where
    E {x ↦ T(Fix(Param(x,t),e),env), env} ⊢ e → v

  E env ⊢ Let(x, t, e1, e2) → v
  where E {x ↦ T(e1, env), env} ⊢ e2 → v

rules // arithmetic

  Num(i) → I(i)

  Ifz(e1, e2, e3) → v
  where e1 → I(i), i = 0, e2 → v

  Ifz(e1, e2, e3) → v
  where e1 → I(i), i ≠ 0, e3 → v

  Add(e1, e2) → I(addInt(i, j))
  where e1 → I(i), e2 → I(j)

  Sub(e1, e2) → I(subInt(i, j))
  where e1 → I(i), e2 → I(j)

  Mul(e1, e2) → I(mulInt(i, j))
  where e1 → I(i), e2 → I(j)
```

Figure 7. Big-step operational semantics for PCF in DynSem.

ular programming language, but usually the semantics of a language is defined only by means of code generation. This makes it difficult to verify that the implementation matches the intended design.

A formal definition of the dynamic semantics enables formal reasoning about the behavior of programs, for example proving that a particular program terminates on a given input, or on all inputs, or that it yields the correct results. In combination with formal definitions of other language concerns, it enables formal reasoning about properties of all programs of the language. For example, we may prove that all well-typed programs run without run-time type errors (type soundness). The problem with formal semantics specifications is that they provide little *direct* value to language designers in terms of providing useful artifacts for their users.

The CompCert compiler [32] defines the dynamic semantics of its source, intermediate and target languages by means of inductive relations in Coq in order to verify the semantics preservation of translations. Specification languages such as Ott [46] provide a more readable language for the production of definitions in proof assistants such as Coq. Semantics engineering frameworks such as Redex [17] and K [15] support definition of executable semantics, which can be used as a basis for testing during language development, but the reduction strategy does not produce efficient interpreters for program development.

We are developing DynSem, a meta-language for the definition of dynamic semantics, that should enable formal reasoning about the behavior of programs in the workbench. However, in order to make defining the semantics worth the while of the language designer, we also aim to generate efficient interpreters from such specifications. Interpreters should at least be usable during software development to avoid the overhead of compilation, and could potentially serve as the sole implementation of a language.

Our current design of DynSem is based on the framework of implicitly-modular structural operational semantics (I-MSOS) developed by Mosses et al. [36, 9]. Fig. 7 defines the DynSem specification for PCF. The rules are similar to conventional big-step structural operational semantics rules. However, due to the I-MSOS approach, semantic components such as environments (and stores, etc.) do not have to be mentioned in rules that do not access these components. For example, the rules for arithmetic in Fig. 7 do not refer to the environment. In such rules, the environment is implicitly passed from conclusion to premises (and a store would be passed between premises from left to right). While we are using DynSem here to define big-step operational semantics, the formalism can also be used for the definition of small-step operational semantics.

5. Deriving Front-Ends

To support programmers in editing programs in a language, the language workbench generates an Eclipse plugin with language-aware editor services that provide interactive feedback about a program during editing. Such feedback includes inline error messages about inconsistencies in a program that appear as you type, project navigation, and contextual discovery. Realizing such interactive services requires analyses to support error recovery in order to not break down on the first error. Furthermore, analysis needs to be incremental so that the effort of analysis is proportional to the changes made. Supporting such features hugely complicates the manual implementation of analyses. By abstracting from implementation strategies in our meta-languages, we can generically forge additional features such as error recovery and incremental analysis on generated implementations.

Spoofax extends Eclipse with an interface for connecting syntactic and semantic editor services [26]. In the process of the project outlined in this paper, we are replacing manually written implementations of editor services by implementations generated from higher-level meta-languages. Generation of syntactic editor services is previous work [26]. We have extended our previous work [51] on incremental name analysis to type analysis.

Syntactic editor services. A declarative syntax definition formalism is the ultimate multi-purpose DSL. From an SDF3 syntax definition we generate a wide range of syntactic editor services. All services rely on the abstract syntax tree schema that is derived from a syntax definition. We generate a parser based on Scannerless Generalized-LR parsing [48, 47] from an SDF3 definition. The resulting parser supports error recovery [13], which allows an AST to be created even if the parsed program is not completely syntactically correct. This means that other editor services relying on an AST can continue to operate in the face of syntactic errors. In addition to deriving a parser, we derive syntax highlighting, code folding rules, and outline views [26]. The template nature of SDF3 enables the generation of syntactic completion templates and pretty-print rules [49].

Semantic editor services. The semantic editor services are built around the name and type analysis algorithm derived from an NaBL and TS definition. This algorithm identifies definitions of names, resolves references to identifiers, computes the types of expressions, and generates error messages in case of failures. The algorithm is incremental due to the use of a 'task engine' that caches resolution tasks and their results [51]. We have introduced the TS language in order to generate task generation rules for type analysis. The results of name and type analysis are used for navigation through reference resolution; for semantic code completion by suggesting possible valid completions of an identifier; and to display error messages, such as for unresolved references, duplicate definitions, and type errors.

6. Deriving Interpreters

To support the direct execution of programs in the object language, we systematically generate a Java-based AST interpreter from a DynSem specification for the language. Each constructor of the language is translated to a class in Java with an `evaluate` method based on the DynSem rules for the constructor.

We choose to generate AST interpreters rather than bytecode interpreters. Bytecode interpreters require a translation to bytecode, while we are aiming at an expression of the semantics at the source level. AST interpreters are simpler to reason about, preserve the structure of the program to be executed and allow optimizations to be performed locally in the program. The AST interpreters that we generate look and act similar to regular Java programs which allows the Java VM to recognize program patterns and perform optimizations. Graal [53], a variation of the Java Hotspot VM, promises to eliminate the overhead of dynamic dispatch in AST interpreters by aggressively inlining program nodes.

```
Explication:

  Add(e1, e2) → I(AddInt(i, j))
  where e1 → I(i), e2 → I(j)

after:

  E env ⊢ Add(e1, e2) → I(AddInt(i, j))
  where E env ⊢ e1 → I(i), E env ⊢ e2 → I(j)
```

```
Factorization:

  E env ⊢ Fun(Param(x, t), e) → C(x, e, env)

after:

  E env ⊢ Fun(p, e) → c
  where p ⇒ Param(x, t), C(x, e, env) ⇒ c
```

```
Merging:

  Ifz(e1, e2, e3) → v
  where e1 → I(i), i ≡ 0, e2 → v

  Ifz(e1, e2, e3) → v
  where e1 → I(i), i ≢ 0, e3 → v

after:

  Ifz(e1, e2, e3) → v
  where e1 → I(i),
    [i ≡ 0, e2 → v] + [i ≢ 0, e3 → v]
```

Figure 8. Examples of transformations on DynSem rules before and after explication, factorization, and merging.

6.1 Rule preprocessing

To simplify code generation we first simplify rules and their premises by explication, factorization, and merging. We explain the individual preprocessing steps below and refer to the examples in Fig. 8.

Explication. DynSem rules implicitly pass the environment to premises. For example, in the `Add` reduction rule in Fig. 8 the unchanged environment is implicitly passed to the evaluation of the left and right hand sides of the expression. Explication rewrites rules so that all environments are explicitly mentioned and passed.

Factorization. We factorize terms to lift nested pattern matches and instantiations to a sequence of premises that only perform simple pattern matches and instantiations. For example, in the rule evaluating `Fun` to closures we lift the nested pattern match on `Param` to a premise.

Rule merging. Evaluation alternatives are expressed in DynSem by multiple rules matching on the same constructor. For example, the DynSem rules for PCF's `ifz` expression from Fig. 8 are overloaded on the matching pattern.

The generator merges these rules into a single one. The first premise of the two rules are identical. Our generator merges overloaded rules and shares the longest matching sequence of premises between the alternatives. In Fig. 8 we use `+` to denote alternatives between premises. This rewriting improves evaluation efficiency by eliminating the redundant evaluation of expression `e1`, the redundant pattern matching against constructor `I(i)`, and the double binding of variable `i`. Merging also eliminates the need for backtracking, which would harm performance due to redundant evaluations and exception raising and catching.

Our factorization and merging technique is similar to left-factoring [3, 39]. The merging strategy does not insert additional rules or constructors to encapsulate alternatives, which would impact performance by increasing the number of dispatch operations and the depth of the evaluation stack.

6.2 Interpreter generation

We transform DynSem rules to an AST interpreter in Java. An AST interpreter represents the running program as an abstract syntax tree, and uses the stack of the host language to store the execution state. Each node of the tree has an `evaluate` method. Fig. 9 shows the interpreter class for the `Ifz` constructor of PCF. Execution begins by invoking the `evaluate` method on some node of the AST; in the PCF case, this is the root node. The `evaluate` code for each node defines the execution of the program subtree rooted at that node; typically this involves invoking the evaluation functions of child nodes. Control is returned to the caller when evaluation has completed successfully. The interpreter aborts evaluation of a program if an error occurs.

Each call to the `evaluate` method passes an evaluation environment (frame) that maps keys to values. The PCF interpreter uses the environment to map variable names to integer values, closures and thunks. Environments are immutable and have hiding semantics on update. This means that evaluation of a child node cannot alter the contents of an environment visible to its parent.

Code generation. The generator translates explicated, factorized, merged, and analyzed rules to evaluation methods. For example, Fig. 9 shows the generated interpreter node for the rule

```
E env ⊢ Ifz(e1, e2, e3) → v
where E env ⊢ e1 → I(i),
      [i ≡ 0, E env ⊢ e2 → v] +
      [i ≢ 0, E env ⊢ e3 → v]
```

Evaluation of the `Ifz` node begins by declaring and binding variables `env`, `e1`, `e2` and `e3`. The interpreter then evaluates the condition `e1` to intermediate value `v1`. The pattern match against variable `v1` is translated to an `instanceof` check and all remaining premise evaluations are nested in the success branch of the match. Binding of variable `i` occurs only if the match succeeds. Premise alternatives from the merged rule are translated to nested `if−then−else` clauses such

```
package org.metaborg.lang.pcf.interpreter.nodes;

public class Ifz_3_Node extends AbstractNode
                        implements I_Exp
{
  public I_Exp _1, _2, _3;

  @Override
  public Value evaluate(I_InterpreterFrame frame){
    I_InterpreterFrame env = frame;
    I_Exp e1 = this._1;
    I_Exp e2 = this._2;
    I_Exp e3 = this._3;
    Value v1 = e1.evaluate(env);
    if (v1 instanceof I_1_Node) {
      I_1_Node c_0 = (I_1_Node) v1;
      int i = c_0._1;
      if (i != 0) {
        return e3.evaluate(env);
      } else {
        if (i == 0) {
          return e2.evaluate(env);
        } else {
          throw new
          InterpreterException("Premise failed");
        }
      }
    } else {
      throw new
      InterpreterException("Premise failed");
    }
  }
  // constructor omitted
}
```

Figure 9. Derived interpreter for PCF's Ifz expression.

that evaluation of `e2` or `e3` only takes place if its guard succeeds. The interpreter throws an exception and evaluation is aborted if a premise with no alternatives fails.

6.3 Results and Future work

The interpreters we generate are reasonably fast. In an interpreter for the Green-Marl graph analysis DSL [22]—implemented manually, but following the same approach—programs evaluate 10 times slower than their compiled equivalents, on average. Over 80% of the effort is spent in environment lookups. A PCF program which recursively computes the sum of the first 500 natural numbers executes on average in 14 ms.[4] This program also spends most of its effort doing environment lookups. We experimented with adding inline caching to variable lookups to capitalize on the stability of the environments. This optimization decreased evaluation time of the recursive PCF program to 0.6 ms. This is a naive example but it suggests that immutable environments together with the local stability of execution in an AST interpreter offer good opportunities for optimizations by inlining caching.

In the future we will investigate both static optimizations of the DynSem rules and dynamic optimizations on the program AST. Static optimizations will include unused variable

[4] On a machine with a 2.7 GHz Intel Core i7 process and 16 GB of RAM

elimination, common premise elimination and mechanized construction of more efficient pattern matching DFAs. The latter two optimizations can significantly reduce the amount of evaluation work required by avoiding redundant evaluations. We believe that much efficiency can be gained by performing program optimization at runtime. Some of the optimizations we envision are premise reordering, inline caching of premise evaluation results, dispatch caching, and guarded program tree rewriting speculating on stable local behavior. The idea of self-optimizing AST interpreters is promoted by the Truffle [53] interpreter framework and the FastR project [25]. We focus our investigation on automatically deriving such optimizations from the DynSem rules.

AST interpreters negatively impact performance due to deep stack allocations during evaluation. Local inlining of evaluation functions is expected to compensate this overhead. Static analysis of the semantic rules can detect some inlining opportunities. The statically detectable cases can be inlined during interpreter generations. Many inlining opportunities can only be detected at runtime. A possibility for these cases is to rely on a new-generation JIT compiler for the JVM – Graal [53]. Graal aggressively inlines stable nodes to reduce function calls. Interpreter developers are expected to explicitly annotate methods that may not be inlined. While not all inlining opportunities are visible statically, many cases where inlining may not be performed are visible statically in the semantics.

7. Deriving Proof Infrastructure

To support verification of properties of language definitions, we systematically generate language models in the Coq proof assistant [10]. The semantics of each meta-language—syntax, name binding, typing, and dynamic semantics—is specified by defining predicates over a simple universal representation of program terms. The semantic predicates describe generic properties of the meta-language, and are parameterized over a further set of definitions and predicates that characterize each particular object language. As an example, the definitions characterizing the PCF language[5] are presented (with some slight simplifications for this paper) in Fig. 11.

To illustrate the approach, we start by presenting the Coq model for the abstract syntax signatures corresponding to syntax definitions in SDF3, which also underlies the models for our other meta-languages.

Syntax Definition An SDF3 definition specifies both the concrete syntax of a particular object language and how syntactically valid programs should be represented as abstract syntax trees. For verification of language properties, we can ignore the concrete syntax of the language (we do not aim at proving parser or pretty-printer correctness at this point), and focus on the abstract syntax. Thus we specify the formal

[5] Available at `https://github.com/metaborgcube/metaborg-pcf`

```
Variable I C : Type.

Inductive term : Type :=
| Co : C → list term → term
| Id : I → term.

Variable S : Type.
Variable sig : C → (list S * S).
Variable I_Sort Prog_Sort : S.

Inductive ws_term : sort → term → Prop :=
| Co_ws cn s ss ts:
   sig cn = (ss,s) →
   Forall2 ws_term ss ts →
   ws_term s (Co cn ts)
| Id_ws x : ws_term I_Sort (Id x).
```

Figure 10. Coq model of SDF3

semantics of an SDF3 definition as the set of trees corresponding to syntactically valid object-language programs.

To capture this idea in Coq, we start by defining a universal term representation suitable for ASTs from arbitrary object languages. Terms are built out of constructor nodes and leaf identifier nodes. The representation is parameterized by a type `I` representing identifiers and inductive set `C` of constructors as presented in Fig. 10.

Not all the terms in `term` make sense for a particular SDF3 definition. For example, the putative PCF term:

```
Co App [Co IntType []]
```

is not well formed because it violates PCF's sort conventions (e.g., constructor `App` has arity 2 and should take two terms of sort `Exp` as arguments whereas `IntType` has sort `Type`). To define the set of *well-sorted* terms, we also extract from the SDF3 definition a set `S` of sorts, a function `sig` mapping each constructor to the sorts of its arguments and its result, a distinguished sort `I_Sort` corresponding to terms representing identifiers and a distinguished sort `Prog_Sort` corresponding to complete programs in this language. Given these parameters, we define in Fig. 10 a well-sortedness predicate `ws_term` $s\ t$ that checks that a particular term t is well sorted for a particular sort s.

Finally, we can define the semantics of the SDF3 definition specified by (`I`, `C`, `S`, `sig`, `I_Sort`, `Prog_Sort`) as the following set of syntactically well-sorted programs:

$$\{t \in \texttt{term} \mid \texttt{ws\_term Prog\_Sort}\ t\}.$$

We chose this approach — starting with a universal term representation and then using predicates to define the well-sorted subset corresponding to a particular SDF3 specification — to let us state and prove generic Coq lemmas that apply across specifications. An alternative approach, which we also plan to explore, would be to generate a specialized term representation (e.g., in which terms are explicitly indexed by sorts) for each SDF3 specification and generate proofs of the generic lemmas as well.

Semantics of Name Binding In a similar style, the semantics of an NaBL name binding specification is given by a

```
(* SDF model *)
Definition I := nat. (* identifiers *)

Inductive C := (* constructors *)
 | ParamC | TNatC | TBoolC | TFunC
 | VarC | LamC | FixC | AppC
 | IfzC | PlusC | TimesC | NatC (n :nat).

Inductive S := (* sorts *)
 | ID_S | Exp_S | Param_S | Type_S .

Definition I_Sort := ID_S.

Fixpoint sig (sc:C) : list S * S :=
  match sc with
    | TFunC ⇒ ([Type_S; Type_S], Type_S)
    | VarC ⇒ ([ID_S],Exp_s)
    | LamC ⇒ ([Param_S; Exp_S],Exp_S)
    | ParamC ⇒ ([ID_S;Type_S], Param_S)
    ...
  end.

Definition Prog_Sort := Exp_S.

Variable (t : term) (R : pos → pos).

(* Nabl model *)
Inductive scopes : term → Prop :=
| Lam_scopes_Var lp : scopes (Co LamC lp)
| Fix_scopes_Var lp : scopes (Co FixC lp).

Inductive defines : term → I → Prop :=
| Param_defines_Var x t :
    defines (Co ParamC ((Id x)::t)) x.

Inductive refers_to : term → I → Prop :=
| Var_refers_Var x lp :
    refers_to (Co VarC ((Id x)::lp)) x.

(* Type system *)
Inductive defines_ty : term → I → term → Prop :=
| Param_types_defines_Var x ty :
    defines_ty (Co ParamC [(Id x);ty]) x ty.

Inductive well_typed : term → term → Prop :=
| VarC_wt x ty :
    lookup x ty →
    well_typed (Co VarC [Id x]) ty
| AppC_wt e1 e2 ty1 ty2 :
    well_typed e1 (Co TFunC [ty1;ty2]) →
    well_typed e2 ty1 →
    well_typed (Co AppC [e1;e2]) ty2
| ...

(* Dynamic semantics *)
Inductive eval : Env → term → value → Prop :=
| VarC_sem x tr env va nenv :
    get_env x env tr nenv →
    eval nenv tr va →
    eval env (Co VarC [Id x]) va
| AppC_sem e1 e2 v e x env nenv :
    eval env e1 (Clos x e nenv) →
    eval { x |−→ (e2,env), nenv} e v →
    eval env (Co AppC [e1;e2]) v
| PlusC_sem e1 e2 env m n:
    eval env e1 (Natv n) →
    eval env e2 (Natv m) →
    eval env (Co PlusC [e1;e2]) (Natv (m+n))
| ...
```

Figure 11. Coq definitions of PCF derived from Spoofax

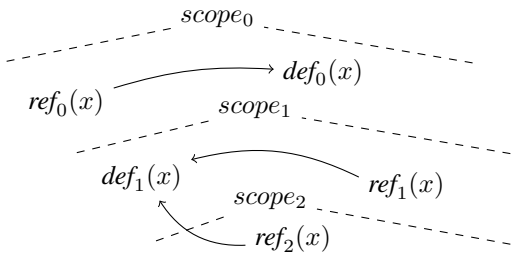

Figure 12. Scoping and resolution

correctness predicate for *resolution maps* over (well-sorted) programs. Resolution is specified in terms of subterm *positions* (of type `pos`) within the top-level program term. A resolution map $\mathtt{R}_t$: `pos` → `pos` for the top-level term t maps each node that represents an identifier use to the node at which the identifier is defined, according to a particular NaBL specification. Rather than defining $\mathtt{R}_t$ constructively, we define a predicate characterizing correctness of $\mathtt{R}_t$ for an arbitrary properly-resolved program t. (To lighten notation, we generally assume a fixed program t and drop the subscript on R.)

The predicate presented here characterizes a substantial subset of the specifications that can be written in NaBL, including mutual recursive definitions, but excluding advanced features such as importat of names from modules. It reflects the following intuitive semantics of resolution, illustrated by Fig. 12.

(i) Each definition *potentially reaches* all nodes dominated by its nearest enclosing scope delimiter. (Note that this permits the scope of definitions to extend upwards and leftwards in the tree as well as rightwards and downwards; this can be useful for, e.g., mutually recursive definitions.) For example, in Fig. 12, the nearest enclosing scope delimiter for def_1 is $scope_1$, so def_1 potentially reaches ref_1 and ref_2. Similarly, def_0 potentially reaches all three references.

(ii) Each reference resolves to the (unique) potentially reaching definition within the *innermost* dominating scope that has such a definition (i.e., definitions in inner scopes shadow those in outer ones). For example, in Fig. 12, ref_1 and ref_2 resolve to def_1 because $scope_1$ lies inside $scope_0$.

Our Coq formalization of these semantics is parameterized by three relations extracted from the binding rules in the NaBL specification (here, for simplicity, we ignore namespaces and some other details involving positions that appear in the actual formalization). `defines` t x holds when term t defines identifier x, `refers_to` t x holds when term t refers to identifier x, and `scopes` t holds when term t delimits a scope. Fig. 11 illustrates the instantiation of these relations for PCF.

$$
\begin{array}{l}
\texttt{scopeof}\ p\ p_s := \\
\quad \exists t_s, t_s@p_s \wedge \texttt{scopes}\ t_s \wedge p_s \prec p \wedge \\
\quad \forall p'_s\ t'_s, \\
\quad\quad (t'_s@p'_s \wedge \texttt{scopes}\ t'_s \wedge p_s \preceq p'_s \prec p) \Longrightarrow p'_s = p_s \\
\\
\texttt{mightreach}\ p_r\ p_d\ p_s := \texttt{scopeof}\ p_d\ p_s \wedge p_s \prec p_r \\
\\
\texttt{reaches}\ p_r\ x\ p_d := \\
\quad \exists t_d, t_d@p_d \wedge \texttt{defines}\ t_d\ x \wedge \\
\quad \exists p_s, \texttt{mightreach}\ p_r\ p_d\ p_s \wedge \\
\quad \forall p'_d\ t'_d\ p'_s,\ t'_d@p'_d \Longrightarrow \\
\quad\quad (\texttt{defines}\ t'_d\ x\ \wedge \texttt{mightreach}\ p_r\ p'_d\ p'_s) \Longrightarrow \\
\quad\quad\quad p'_s \prec p_s \vee (p'_s = p_s \wedge p'_d = p_d) \\
\\
\texttt{sound}\ \texttt{R} := \\
\quad \forall p_r\ p_d, \texttt{R}(p_r) = p_d \Longrightarrow \\
\quad\quad \exists x\ t_r, t_r@p_r \wedge \texttt{refers\_to}\ t_r\ x \wedge \\
\quad\quad \exists t_d, t_d@p_d \wedge \texttt{defines}\ t_d\ x\ \wedge\ \texttt{reaches}\ p_r\ x\ p_d \\
\\
\texttt{complete}\ \texttt{R} := \\
\quad \forall p_r\ t_r\ x, t_r@p_r \wedge \texttt{refers\_to}\ t_r\ x\ \Longrightarrow \\
\quad\quad \exists p_d, \texttt{R}(p_r) = p_d \\
\\
\texttt{lookup}\ x\ p_r\ ty := \\
\quad \exists p_d\ t_d, \texttt{R}(p_r) = p_d \wedge t_d@p_d\ \wedge \texttt{defines\_ty}\ t_d\ x\ ty
\end{array}
$$

Figure 13. NaBL mapping correctness

We now proceed assuming a fixed top-level term representing the whole program; positions implicitly refer to this term. In Fig. 13, we define some useful auxiliary predicates. We write $t@p$ to assert that t is the subterm at position p, and write $p_1 \preceq p_2$ (resp. $p_1 \prec p_2$) to mean that the node at position p_1 is an ancestor (resp. a strict ancestor) of the node at position p_2. The `scopeof` $p\ p_s$ predicate says that the nearest enclosing scope delimiter of the node at position p is at position p_s. The `mightreach` $p_r\ p_d\ p_s$ predicate says that a definition at node p_d potentially reaches a reference at node p_r within the scope delimited by p_s. The `reaches` $p_r\ x\ p_d$ predicate says that the node at p_d defines x and that this definition definitely reaches the node at p_r, because it potentially reaches it and any other definition of x that potentially reaches it is in an outer scope (and is therefore shadowed).

Finally, we can define a correct map `R` as one that is sound and complete, according to the `sound` and `complete` definitions from Fig. 13.

Type System The Coq formalization of the type language TS is largely straightforward. Types are represented as `terms` having distinguished sort `Type_S`. The semantics of a TS specification is given by a Coq inductive relation `well_typed R` $t\ ty$ (also written $\vdash_{\texttt{R}}\ t : ty$) between (program) terms t and types ty, which is generated in an obvious way from the TS type rules. Fig. 11 shows some typical clauses of this relation.

Unlike a conventional typing relation, this one lacks an environment component for looking up the types of variables. Instead, to give meaning to the `where definition of` clause in a TS specification (e.g. for PCF `Var` nodes), we use the type information associated with variable declarations by NaBL specifications.

We extend the `defines` relation introduced previously to include a type component. The `defines_ty` $t\ x\ ty$ predicate holds when the term t defines identifier x with type ty. To determine the type of an identifier at a reference, we use the resolution map `R` to find the corresponding definition, as described by the `lookup` relation in Fig. 13. The `well_typed` clause for `Var` in Fig. 11 illustrates the use of `lookup`.

Dynamic Semantics Translation of the dynamic semantics as defined in DynSem is also quite straightforward. The DynSem constructors declared with sort `value` are used to define a new Coq inductive type `value` and the DynSem definitions are directly translated into an inductive relation `eval` $E\ t\ v$ (also written $E \vdash t \Rightarrow v$) over environments E, terms t and values v.

Values and operations considered "native" to the DynSem specification are directly translated into their existing Coq equivalent. As an example, in PCF, the natural numbers are directly embedded in the constructors `NatC n` and values `Natv n` , and the `PlusC` constructor directly evaluates to the Coq + function.

Type Preservation To illustrate that our Coq models can be used to prove useful properties of language definitions, we have developed (manually) a Coq proof of type preservation for PCF. To do this, we first extend the typing relation $\vdash$ to values in the obvious way. We can then prove:

$$
\begin{array}{l}
\forall e\ \texttt{R}_e\ v\ ty, \\
\quad (\texttt{ws\_term Exp\_S}\ e\ \wedge \texttt{sound}\ \texttt{R}_e \wedge \texttt{complete}\ \texttt{R}_e) \Longrightarrow \\
\quad (\emptyset \vdash e \Rightarrow v \wedge \vdash_{\texttt{R}_e} e : ty) \Longrightarrow \\
\quad\quad \vdash v : ty
\end{array}
$$

In future work, we hope to automate routine proofs of this kind, at least for some languages.

8. Related Work

The language definition landscape provides the following complementary categories of tools for language designers and engineers:

- Language workbenches such as Xtext [54], MPS [50], Rascal [29], and Spoofax [26] support separation of concerns in implementation of compilers and IDEs, but do not support specification of dynamic semantics, verification, or test generation. Erdweg et al. [16] give an overview of the features of modern language workbenches.
- Semantics frameworks such as PLT Redex [17], K [31], and I-MSOS [37, 9] support separation of concerns in specification of dynamic semantics. The main goal is the formalization and documentation of the dynamic semantics of programming languages or models of full-fledged

languages. Specifications may be validated by executing programs (e.g., generated through random testing). However, specifications are not intended to support static verification of language properties or the generation of high performance interpreters.

- Proof assistants such as Coq [10], Isabelle/HOL [23], and Twelf [40] support construction of machine-checked proofs of meta-theoretic properties of languages and implementations. These are general-purpose systems that largely lack specific features for describing programming languages, although Twelf does have primitive support for higher-order abstract syntax. Language encodings tend to be low-level, targeted at a single purpose, and tailored to the particular features of a single prover. Frameworks such as Ott [46] and Lem [38] address these short-comings by providing a single uniform mechanism for defining semantics as transition systems, from which encodings in a variety of provers, executable languages, and typesetting systems can be extracted automatically. However, the verification proper (construction of proofs) is still done using the generated encoding in the language of the proof assistant.

Executable type systems. Formal definitions of type systems are specialized to a particular application. Type systems in semantics engineering take the form of derivation systems, defining judgments of the form $\Gamma \vdash e : t$ — expression e has type t in environment Γ — by means of derivation rules. Such derivation systems target the production of documentation and proofs [52]. Name binding and scope rules are typically incorporated in these systems by propagation of environments. On the other hand, language workbenches such as Xtext [54] and Spoofax [21] separate the concerns of name resolution, type analysis, and error constraints, targeting the production of type checkers for IDEs. The use of general purpose languages (programming languages, term rewriting systems, attribute grammars), complicates the verification of the correctness of such definitions.

A declarative type system abstracts from the algorithmics of type checking and type inference, much like a declarative syntax definition abstracts from the algorithmics of parsing [27]. However, unlike the domain of syntax definition, it is not common practice to *generate* type checkers from type system specifications. Type systems that are convenient for reasoning may be non-deterministic, and need to be massaged for use in a non-backtracking algorithm [34, 42]. An interesting direction for future work will be to investigate the automatic generation of algorithmic type checkers from highly declarative type system specifications.

Executable dynamic semantics. Semantic frameworks target simulation of dynamic semantics specifications, rather than generating efficient interpreters, as is the goal for DynSem. The K semantic framework [31] provides an integrated language for defining the syntax, static, and dynamic semantics of a language. Dynamic semantics are defined in terms of structural and reduction rules. The former transform the structure of a program's data such that the latter rules can be applied to evaluate the program. Specifications in K are compiled to interpreters in Maude. By comparison, in DynSem one only specifies the reduction rules and there are no structural rules. The interpreters we derive can perform rewriting but they do not rely on rewriting for program evaluation. A PCF interpreter derived from a specification in K had an evaluation time for the naive recursive summation program of Sect. 6 in the order of minutes (as opposed to milliseconds for our DynSem interpreter).

Dynamic semantics of languages defined in PLT Redex [17] are given as reduction relations with a reduction context. The reduction context determines the next possible evaluation steps to be performed. In the general case, the search for each step needs to start at the root of the AST, adding to each reduction step a tree traversal, which makes execution expensive. In addition, Redex definitions typically use explicit substitution for binding values to names. This simplifies definitions, but also makes execution expensive, as the application of a substitution requires another traversal of the AST. By comparison, specifications in DynSem use environments as an alternative to explicit substitution.

The DynSem language is based on the I-MSOS approach to operational semantics. As such it can be used for specifications in small-step and big-step style. Small-step semantics definitions are more suitable for verification. However, small-step semantics lead to interpreters with worse performance than those derived from big-step semantics. This is caused by the need to scan the entire intermediate program to discover the next evaluation step to be performed. Small-step semantics can be transformed into equivalent big-step rules by refocusing [12]. Bach Poulsen and Mosses [3] show promising benchmark results for the application of refocusing on a small set of benchmarks.

Big-step style rules lack expressivity for specification of abrupt termination and divergence. Pretty-big-step semantics [7] provides a compact notation for explicitly expressing exceptions and concurrency. Exceptions are accompanied by evaluation traces and are valid outcomes of reduction rules. This eliminates the need for implicit backtracking which is non-trivial in the presence of side-effects. Bach Poulsen and Mosses [4] demonstrate the derivation of pretty-big-step semantics from small-step rules. We plan to explore the advantages and disadvantages of the different styles of operational semantics in DynSem and its interpreter generator.

Combining specification, implementation and verification. The goal of the Language Designer's Workbench project described in this paper is to combine the features of each of these categories of tools into a workbench that provides comprehensive support for all aspects of language design, implementation, and validation. These features have not often

been integrated in prior work, particularly not within an extensible framework.

The CompCert [32] C compiler specifies language semantics, compiler implementation, and compiler properties all within the Coq proof assistant, which uses a single language to express specifications, programs, and theorems connecting the two (although the proofs of these theorems typically rely on a separate tactic language). This approach relies heavily on Coq's ability to extract programs into an efficiently compilable language such as OCaml. However, CompCert is not intended as a general framework for constructing verified compilers, although it does contain a number of supporting libraries that could be useful for compiling source languages other than C.

The 3MT theory of Delaware et al. [14] describes a modular framework in which various effectful language features can be individually specified, together with proofs of their type soundness, and then freely composed to form different languages in such a way that the proofs compose too. Although 3MT's current implementation in Coq requires a rather heavy encoding, we expect its insights about proof modularization to prove valuable in our framework.

Lorenzen and Erdweg [33] describe a system for automatically verifying soundness of typing rules for macro extensions, given desugaring transformations into a type-safe base language. In essence, they verify that the macro typing rules are logically implied by the base language rules applied to the desugared code. Given a suitably rich base language, this technique might be used to verify type soundness for many useful domain-specific language features.

9. Conclusion

In this paper, we have presented the vision of a language *designer's* workbench, which reduces the effort of language design by integrating implementation and verification. Our proof of concept suggests that this vision is at least feasible for a class of simple languages, and provides an outlook on a future in which software language design can be a matter of exploring design alternatives rather than grinding away on tedious implementation details. Realizing this vision requires research in three directions. We must: (1) extend the coverage of our meta-languages to support a larger range of languages; (2) improve performance of analysis algorithms and interpreters to reach production quality implementations; and (3) provide highly automated support for verification of language definitions.

These directions are not independent, as there is a trade-off between expressivity (coverage) of our meta-languages and the extent to which implementation and verification can be automated. Semantics frameworks target semantics engineers, with or working towards a PhD in programming language theory, designing advanced programming languages with complex type systems. Language workbenches target software engineers, with a master's degree in computer science, designing DSLs to automate software development in a particular domain. This difference justifies a trade-off, increasing automation of verification at the expense of expressivity, in order to help software engineers avoid language design pitfalls.

Acknowledgments

We thank Peter Mosses and Martin Churchill for hosting the productive visit of Eelco Visser to Swansea that led to the DynSem language. We thank Peter Mosses, Casper Bach Poulsen, Paolo Torrini, and Neil Sculthorpe for comments on a draft of this paper. We thank the participants of the Summer School on Language Frameworks in Sinaia, Romania in July 2012 — in particular Grigore Rosu, Robby Findler, and Peter Mosses — for inspiring us to address dynamic semantics in language workbenches. This research was partially funded by the NWO VICI *Language Designer's Workbench* project (639.023.206), the NWO Free Competition *Language Libraries* project (612.001.114), and by a gift from the Oracle Corporation.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] B. E. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In G. C. Necula and P. Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 3–15. ACM, 2008.

[3] C. Bach Poulsen and P. D. Mosses. Generating specialized interpreters for modular structural operational semantics. In *Proceedings of the 23rd international symposium on Logic Based Program Synthesis and Transformation*, LOPSTR, 2013.

[4] C. Bach Poulsen and P. D. Mosses. Deriving pretty-big-step semantics from small-step semantics. In Z. Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 270–289. Springer, 2014.

[5] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69(1-2):123–178, 2006.

[6] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[7] A. Charguéraud. Pretty-big-step semantics. In M. Felleisen and P. Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy,*

March 16-24, 2013. Proceedings, volume 7792 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2013.

[8] A. J. Chlipala. A verified compiler for an impure functional language. In M. V. Hermenegildo and J. Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 93–106. ACM, 2010.

[9] M. Churchill, P. D. Mosses, and P. Torrini. Reusable components of semantic specifications. In W. Binder, E. Ernst, A. Peternier, and R. Hirschfeld, editors, *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*, pages 145–156. ACM, 2014.

[10] Coq development team. The Coq proof assistant reference manual. `http://coq.inria.fr`, 2012. Version 8.4.

[11] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In I. Crnkovic and A. Bertolino, editors, *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 185–194. ACM, 2007.

[12] O. Danvy and L. R. Nielsen. Refocusing in reduction semantics. Technical Report RS-04-26, BRICS, Department of Computer Science, University of Aarhus, Nov. 2004.

[13] M. de Jonge, L. C. L. Kats, E. Visser, and E. Söderberg. Natural and flexible error recovery for generated modular language environments. *ACM Transactions on Programming Languages and Systems*, 34(4):15, 2012.

[14] B. Delaware, S. Keuchel, T. Schrijvers, and B. C. Oliveira. Modular monadic meta-theory. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 319–330. ACM, 2013.

[15] C. Ellison and G. Rosu. An executable formal semantics of C with applications. In J. Field and M. Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 533–544. ACM, 2012.

[16] S. Erdweg, T. van der Storm, M. Völter, et al. The state of the art in language workbenches - conclusions from the language workbench challenge. In M. Erwig, R. F. Paige, and E. V. Wyk, editors, *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, volume 8225 of *Lecture Notes in Computer Science*, pages 197–217. Springer, 2013.

[17] M. Felleisen, R. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[18] M. Fowler. Language workbenches: The killer-app for domain specific languages?, 2005.

[19] G. Hedin and E. Magnusson. Jastadd–an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.

[20] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[21] Z. Hemel, L. C. L. Kats, D. M. Groenewegen, and E. Visser. Code generation by model transformation: a case study in transformation modularity. *Software and Systems Modeling*, 9(3):375–402, 2010.

[22] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: a DSL for easy and efficient graph analysis. In T. Harris and M. L. Scott, editors, *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, pages 349–362. ACM, 2012.

[23] Isabelle2013-2 tutorials and manuals. `http://www.cl.cam.ac.uk/research/hvg/Isabelle/documentation.html`, 2013.

[24] JetBrains. Meta programming system. `https://www.jetbrains.com/mps`.

[25] T. Kalibera, P. Maj, F. Morandat, and J. Vitek. A fast abstract syntax tree interpreter for R. In M. Hirzel, E. Petrank, and D. Tsafrir, editors, *10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '14, Salt Lake City, UT, USA, March 01 - 02, 2014*, pages 89–102. ACM, 2014.

[26] L. C. L. Kats and E. Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463, Reno/Tahoe, Nevada, 2010. ACM.

[27] L. C. L. Kats, E. Visser, and G. Wachsmuth. Pure and declarative syntax definition: paradise lost and regained. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 918–932, Reno/Tahoe, Nevada, 2010. ACM.

[28] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler. Run your research: on the effectiveness of lightweight mechanization. In J. Field and M. Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 285–296. ACM, 2012.

[29] P. Klint, T. van der Storm, and J. J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*, pages 168–177. IEEE Computer Society, 2009.

[30] G. D. P. Konat, L. C. L. Kats, G. Wachsmuth, and E. Visser. Declarative name binding and scope rules. In K. Czarnecki and G. Hedin, editors, *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012*, volume 7745 of *Lecture Notes in Computer Science*, pages 311–331. Springer, 2012.

[31] D. Lazar, A. Arusoaie, T. F. Serbanuta, C. Ellison, R. Mereuta, D. Lucanu, and G. Rosu. Executing formal semantics with the K tool. In D. Giannakopoulou and D. Méry, editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 267–271. Springer, 2012.

[32] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[33] F. Lorenzen and S. Erdweg. Modular and automated type-soundness verification for language extensions. In G. Morrisett and T. Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 331–342. ACM, 2013.

[34] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[35] J. C. Mitchell. *Foundations for programming languages*. Foundation of computing series. MIT Press, 1996.

[36] P. D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:195–228, 2004.

[37] P. D. Mosses and M. J. New. Implicit propagation in structural operational semantics. *Electronic Notes in Theoretical Computer Science*, 229(4):49–66, 2009.

[38] S. Owens, P. Böhm, F. Z. Nardelli, and P. Sewell. Lem: A lightweight tool for heavyweight semantics. In M. C. J. D. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, editors, *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings*, volume 6898 of *Lecture Notes in Computer Science*, pages 363–369. Springer, 2011.

[39] M. Pettersson. A compiler for natural semantics. In T. Gyimóthy, editor, *Compiler Construction, 6th International Conference, CC 96, Linköping, Sweden, April 24-26, 1996, Proceedings*, volume 1060 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 1996.

[40] F. Pfenning and C. Schürmann. Twelf user's guide, version 1.4. `http://www.cs.cmu.edu/~twelf/guide-1-4`, 2002.

[41] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, Massachusetts, 2002.

[42] B. C. Pierce and D. N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, 2000.

[43] M. Roberson, M. Harries, P. T. Darga, and C. Boyapati. Efficient software model checking of soundness of type systems. In G. E. Harris, editor, *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 493–504. ACM, 2008.

[44] G. Rosu and T.-F. Serbanuta. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[45] M. Schäfer, T. Ekman, and O. de Moor. Challenge proposal: verification of refactorings. In T. Altenkirch and T. D. Millstein, editors, *Proceedings of the 3rd ACM Workshop Programming Languages meets Program Verification, PLPV 2009, Savannah, GA, USA, January 20, 2009*, pages 67–72. ACM, 2009.

[46] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1):71–122, 2010.

[47] M. G. J. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In R. N. Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158. Springer, 2002.

[48] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[49] T. Vollebregt, L. C. L. Kats, and E. Visser. Declarative specification of template-based textual editors. In A. Sloane and S. Andova, editors, *International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012*, page 8. ACM, 2012.

[50] M. Völter and K. Solomatov. Language modularization and composition with projectional language workbenches illustrated with MPS. In M. G. J. van den Brand, B. Malloy, and S. Staab, editors, *Software Language Engineering, Third International Conference, SLE 2010*, Lecture Notes in Computer Science. Springer, 2010.

[51] G. Wachsmuth, G. D. P. Konat, V. A. Vergu, D. M. Groenewegen, and E. Visser. A language independent task engine for incremental name and type analysis. In M. Erwig, R. F. Paige, and E. V. Wyk, editors, *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, volume 8225 of *Lecture Notes in Computer Science*, pages 260–280. Springer, 2013.

[52] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, November 1994.

[53] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In A. L. Hosking, P. T. Eugster, and R. Hirschfeld, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, pages 187–204. ACM, 2013.

[54] Xtext documentation. `http://www.eclipse.org/Xtext/documentation/2.6.0/Xtext`, 2014.

[55] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In M. W. Hall and D. A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294. ACM, 2011.