A Language Independent Task Engine for Incremental Name and Type Analysis

Guido H. Wachsmuth[1,2], Gabriël D.P. Konat[1], Vlad A. Vergu[1], Danny M. Groenewegen[1], and Eelco Visser[1]

[1] Delft University of Technology, The Netherlands
{g.h.wachsmuth,v.a.vergu,d.m.groenewegen}@tudelft.nl,
{gkonat,visser}@acm.org
[2] Oracle Labs, Redwood City, CA, USA

Abstract. IDEs depend on incremental name and type analysis for responsive feedback for large projects. In this paper, we present a language-independent approach for incremental name and type analysis. Analysis consists of two phases. The first phase analyzes lexical scopes and binding instances and creates deferred analysis tasks. A task captures a single name resolution or type analysis step. Tasks might depend on other tasks and are evaluated in the second phase. Incrementality is supported on file and task level. When a file changes, only this file is recollected and only those tasks are reevaluated, which are affected by the changes in the collected data. The analysis does neither re-parse nor re-traverse unchanged files, even if they are affected by changes in other files. We implemented the approach as part of the Spoofax Language Workbench and evaluated it for the WebDSL web programming language.

1 Introduction

Integrated development environments (IDEs) provide a wide variety of language-specific editor services such as syntax highlighting, error marking, code navigation, content completion, and outline views in real-time, while a program is edited. These services require syntactic and semantic analyses of the edited program. Thereby, timely availability of analysis results is essential for IDE responsiveness. Whole-program analyses do not scale because the size of the program determines the performance of such analyses.

Incremental analysis reuses previous analysis results of unchanged program parts and reanalyses only parts affected by changes. The granularity of the incremental analysis directly impacts the performance of the analysis. A more fine-grained incremental analysis is able to reanalyze smaller units of change, but requires a more complex change and dependency analysis. At program level, any change requires reanalysis of the entire program, which might consider the results of the previous analysis. At file level, a file change requires reanalysis of the entire file and all dependent files. At program element level, changes to an element within a file require reanalysis of that element and dependent elements, but typically not of entire files. Incremental analyses are typically implemented

M. Erwig, R.F. Paige, and E. Van Wyk (Eds.): SLE 2013, LNCS 8225, pp. 260–280, 2013.

manually. Thereby, change detection and dependency tracking are cross-cutting the implementation of the actual analysis. This raises complexity of the implementation and negatively affects maintenance, reusability, and modularity.

In this paper, we focus on incremental name and type analysis. We present a language-independent approach which consists of two phases. The first phase analyzes lexical scopes, collects information about binding instances, and creates deferred *analysis tasks* in a top-down traversal. An analysis task captures a single name resolution or type analysis step. Tasks might depend on other tasks and are evaluated in the second phase. Incrementality is supported on file level by the collection phase and on task level by the evaluation phase. When a file changes, only this file is recollected and only those tasks are reevaluated, which are affected by the changes in the collected data. As a consequence, the analysis does neither re-parse nor re-traverse unchanged files, even if they are affected by changes in other files. Only the affected analysis tasks are reevaluated.

Our approach enables language engineers to abstract over incrementality. When applied directly, language engineers need to parametrize the collection phase, where they have full freedom to create and combine low-level analysis tasks. Thereby, they can focus solely on the name binding and typing rules of their language while the generic evaluation phase provides the incrementality. The approach can also form the basis for more high-level meta-languages for specifying the static semantics of programming languages. We use the task engine to implement incremental name analysis for name binding and scope rules expressed in NaBL, Spoofax' declarative name binding language [16].

We have implemented the approach as part of the Spoofax language workbench [14] and evaluated it for WebDSL, a domain-specific language for the implementation of dynamic web applications [7], designed specifically to enable static analysis and cross-aspect consistency checking in mind [11]. We used real change-sets from the histories of two WebDSL applications to drive experiments for the evaluation of the correctness, performance and scalability of the obtained incremental static analysis. Experiment input data and the obtained results are publicly available.

We proceed as follows. In the next section, we introduce the basics of name and type analysis and introduce the running example of the paper. In Sects. 3 and 4, we discuss the two analysis phases of our approach, collection and evaluation. In Sect. 5, we discuss the implementation and its integration into the Spoofax language workbench. In Sect. 6, we discuss the evaluation of our approach. Sects. 7 and 8 are for related work and conclusions.

2 Name and Type Analysis

In this section, we discuss name and type analysis in the context of the running example of the paper, a multi-file C# program shown in Fig. 1.

Name Analysis. In textual programming languages, an *identifier* is a name given to program elements such as variables, methods, classes, and packages. The

```
class A {
  B b; int m;
  float m() {
    return 1 + b.f; }}
```

```
class B {
  int i; float f;
  int m() {
    return 0; }}
```

```
class C:A {

  int n() {
    return m(); }}
```

Fig. 1. C# class declarations in separate files with cross-file references. The underlined expression causes a type error.

```
class A {
  B b; int m;
  int m(B b) {
    return 1 + b.i; }}
```

```
class B {
  int i; float f;
  int m() {
    return 1; }}
```

```
namespace N {
  class C:B {
    int n() {
      return m(); }}}
```

Fig. 2. C# class declarations after editing. Changes w.r.t. Fig. 1 are highlighted.

same identifier can have multiple *instances* in different places in a program. Name analysis establishes relations between a *binding instance* that *defines* a name and a *bound instance* that *uses* that name [17]. Name analysis is typically defined programmatically through a name resolution algorithm that connects *binding prospects* to binding instances. When a prospect is successfully connected, it becomes a bound instance. Otherwise, it is a *free instance*.

The C# class declarations in Fig. 1 contain several references, some of which cross file boundaries. The declared type of field `b` in class `A` refers to class `B` in a separate file. Also, the return expression of method `m` in class `A` accesses field `f` in class `B`. The parent of class `C` refers to class `A` in a separate file and the return expression of method `n` in class `C` is a call to method `m` in class `A`.

Languages typically distinguish several *namespaces*, i.e. different kinds of names, such that an occurrence of a name in one namespace is not related to an occurrence of that same name in another. In the example, class `A` contains a field and a homonym method `m`, but C# distinguishes field and method names.

Scopes restrict the visibility of binding instances. They can be nested and name analysis typically looks for binding instances from inner to outer scopes. In the example, `b` is resolved by first looking for a variable `b` in method `A.m`, before looking for a field `b` in class `A`. A *named scope* is the context for a binding instance, and scopes other binding instances. In the example, class `A` is a named scope. It is the context for a class name and a scope for method and field names.

An *alias* introduces a new binding instance for an already existing one. An *import* introduces binding instances from one scope into another one. In the example, class `C` imports fields and methods from its parent class `A`.

Type Analysis. In statically typed programming languages, a *type* classifies program elements such as expressions according to the kind of values they compute [20]. Fig. 1 declares method `C.n` of type **`int`**, meaning that this method is expected to compute signed 32-bit integer values. Type analysis assigns types to program elements. Types are typically calculated compositionally, with the type of a program element depending only on the types of its sub-elements [20].

Type checking compares expected with actual types of program elements. A *type error* occurs if actual and expected type are incompatible. Type errors reveal at compile-time certain kinds of program misbehavior at run-time. In the example, the return expression in method `C.n` causes a type error. The expression is of type **`float`**, since the called method `m` returns values of this type. But the declaration of `C.n` states that it evaluates to values of type **`int`**.

Incremental Analysis. When a program changes, it needs to be reanalyzed. Different kinds of changes influence name and type analysis. First, adding a binding instance may introduce bindings for free instances, or rebind bound instances. Removing a binding instance influences all its bound instances, which are either rebound to other binding instances or become free instances. Changing a binding instance combines the effects of removing and adding. Second, adding a binding prospect requires resolution, while removing it makes a binding obsolete. Changing a binding prospect requires re-binding, resulting either in a new binding or a free instance. Third, addition, removal, or change of scopes or imports influence bound instances in the affected scopes, which might be rebound to different binding instances or become free instances. Similarly, they influence bound instances which are bound to binding instances in the affected scopes. Finally, addition of a typed element requires type analysis, while removing it makes a type calculation obsolete. Changing a typed element requires reanalysis.

Furthermore, changes propagate along dependencies. When bound instances are rebound to different binding instances or become free instances, this influences bindings in the context of these bound instances, the type of these instances, the type of enclosing program elements, and bindings in the context of such types. Consider Fig. 2 for an example. It shows edited versions of the C# class declarations from Fig. 1. We assume the following editing sequence:

1. The return type of method `A.m` is changed from **`float`** to **`int`**. This affects the type of the return expression of method `C.n` and solves the type error, but raises a new type error in the return expression of `A.m`.
2. The return expression of method `A.m` is changed to `b.i`. This requires resolution of `i` and affects the type of the expression, solving the type error.
3. Parameter `B b` is added to method `A.m`. This might affect the resolution and by this the type of `b` and `i` in the return expression, the type of the return expression, the resolution of `m` in method `C.n`, and the type of its return expression. Actually, only the resolution of `b` and `m` and the type of the return expression in `C.n` are affected. The latter resolution fails, causing a resolution error and leaving the return expression untyped.
4. The parent of class `C` is changed from `A` to `B`. This affects the resolution of `m` in method `C.n` and the type of its return expression. It fixes the resolution error and the return expression becomes typed again.
5. Class `C` is enclosed in a new namespace `N`. This might affect the resolution of parent class `B`, the resolution of `m` in `N.C.n`, and the type of the return expression in `N.C.n`. Actually, it does not affect any of those.
6. The return expression of method `m` in class `B` is changed. This might affect the type of this expression, but actually it does not.

We discuss incremental analysis in the next sections. We start with the collection phase in Sect. 3, and continue with the evaluation phase in Sect. 4.

3 Semantic Index

We collect name binding information for all units in a project into a *semantic index*, a central data structure that is persisted across invocations of the analysis and across editing sessions. For the purpose of this paper, we model this data structure as binary relations over keys and values. As keys, we use URIs, which identify bindings uniquely across a project. As values, we use either URIs or terms. We use $\mathcal{U}$ and $\mathcal{T}$ to denote the set of all URIs and terms, respectively.

URIs. We assign a URI to each binding instance, bound instance, and free instance. A bound instance shares the URI with its corresponding binding instance. A URI consists of a language name, a list of scope segments, the namespace of the instance, its name, and an optional unique qualifier. This qualifier helps to distinguish unique binding instances by numbering them consecutively. A segment for a named scope consists of the namespace, the name, and the qualifier of the scoping binding instance. Anonymous scopes are represented by a segment `anon(u)`, where `u` is a unique string to distinguish different scopes. For example, `C#://Class.A.1/Method.m.1` identifies method `m` in class `A` in the C# program in Fig. 1. The qualifier `1` distinguishes the method. Possible homonym methods in the same class would get subsequent qualifiers.

Index Entries. The index stores binding instances ($B \subseteq \mathcal{U} \times \mathcal{U}$), aliases ($A \subseteq \mathcal{U} \times \mathcal{U}$), transitive and non-transitive imports for each namespace ns ($TI_{ns} \subseteq \mathcal{U} \times \mathcal{U}$ and $NI_{ns} \subseteq \mathcal{U} \times \mathcal{U}$), and types of binding instances ($P_{type} \subseteq \mathcal{U} \times \mathcal{T}$). For a binding instance with URI u, B contains an entry (u', u), where u' is retrieved from u by omitting the unique qualifier. u' is useful to resolve binding prospects, as we will show later. An alias consists of the new name, that is a binding instance, and the old name, that is a binding prospect. For each alias, A contains an entry (a, u), where a is the URI of the binding instance and u is the URI of the binding prospect. For a transitive wildcard import from a scope with URI u into a scope with URI u', TI_{ns} contains an entry (u', u). Similarly, $NI_{\mathtt{ns}}$ contains entries for non-transitive imports. Finally, for a binding instance of URI u and of type t, P_{type} contains an entry (u, t). P can also store other properties of binding instances, but we focus on types for this paper.

Example. Fig. 3 shows the index for the running example. It contains entries in B for binding instances of classes `A`, `B`, and `C`, fields `A.b`, `A.m`, `B.i`, and `B.f`, and methods `A.m`, `B.m`, and `C.n`. Corresponding entries for P_{type} contain the types of all fields and methods in the program. Since the running example does not define any aliases, A does not contain any entries. It also contains corresponding entries for NI_{Field}, TI_{Field}, NI_{Method}, and TI_{Method}. These entries model inheritance by a combination of a non-transitive and a transitive import. `C` first inherits the fields and methods from `A` (non-transitive import). Second, `C` inherits the fields and methods which are inherited by `A` (transitive import).

Relation	Key	Value
B	C#:/*Class*.A	C#:/*Class*.A.1
	C#:/*Class*.A.1/*Field*.b	C#:/*Class*.A.1/*Field*.b.1
	C#:/*Class*.A.1/*Field*.m	C#:/*Class*.A.1/*Field*.m.1
	C#:/*Class*.A.1/*Method*.m	C#:/*Class*.A.1/*Method*.m.1
	C#:/*Class*.B	C#:/*Class*.B.1
	C#:/*Class*.B.1/*Field*.i	C#:/*Class*.B.1/*Field*.i.1
	C#:/*Class*.B.1/*Field*.f	C#:/*Class*.B.1/*Field*.i.1
	C#:/*Class*.B.1/*Method*.m	C#:/*Class*.B.1/*Method*.m.1
	C#:/*Class*.C	C#:/*Class*.C.1
	C#:/*Class*.C.1/*Method*.n	C#:/*Class*.C.1/*Method*.n.1
NI_{Field}, TI_{Field}	C#:/*Class*.C.1	Task:/31
NI_{Method}, TI_{Method}	C#:/*Class*.C.1	Task:/31
P_{type}	C#:/*Class*.A.1/*Field*.b.1	Task:/6
	C#:/*Class*.A.1/*Field*.m.1	int
	C#:/*Class*.A.1/*Method*.m.1	([], float)
	C#:/*Class*.B.1/*Field*.i.1	int
	C#:/*Class*.B.1/*Field*.f.1	float)
	C#:/*Class*.B.1/*Method*.m.1	([], int)
	C#:/*Class*.C.1/*Method*.n.1	([], int)

Change	Key	Value
$\Delta^1_{P_{type}}$	C#:/*Class*.A.1/*Method*.m.1	([], float)
	C#:/*Class*.A.1/*Method*.m.1	([], int)
Δ^3_B	C#:/*Class*.A.1/*Method*.m.1/*Var*.b	C#:/*Class*.A.1/*Method*.m.1/*Var*.b.1
$\Delta^3_{P_{type}}$	C#:/*Class*.A.1/*Method*.m.1/*Var*.b.1	Task:/6
	C#:/*Class*.A.1/*Method*.m.1	([], int)
	C#:/*Class*.A.1/*Method*.m.1	([Task:/6], int)
$\Delta^4_{I_{Field}}$	C#:/*Class*.C.1	Task:/31
	C#:/*Class*.C.1	Task:/6
$\Delta^4_{I_{Method}}$	C#:/*Class*.C.1	Task:/31
	C#:/*Class*.C.1	Task:/6
Δ^5_B	C#:/Ns.N	C#:/Ns.N.1
	C#:/*Class*.C	C#:/*Class*.C.1
	C#:/Ns.N.1/*Class*.C	C#:/Ns.N.1/*Class*.C.1
	C#:/*Class*.C.1/*Method*.n	C#:/*Class*.C.1/*Method*.n.1
	C#:/Ns.N.1/*Class*.C.1/*Method*.n	C#:/Ns.N.1/*Class*.C.1/*Method*.n.1
$\Delta^5_{I_{Field}}$	C#:/*Class*.C.1	Task:/6
	C#:/Ns.N.1/*Class*.C.1	Task:/54
$\Delta^5_{I_{Method}}$	C#:/*Class*.C.1	Task:/6
	C#:/Ns.N.1/*Class*.C.1	Task:/54
$\Delta^5_{P_{type}}$	C#:/*Class*.C.1/*Method*.n.1	([], int)
	C#:/Ns.N.1/*Class*.C.1/*Method*.n.1	([], int)

Fig. 3. Initial semantic index for the C# program in Fig. 1 (top) and changes for the C# program from Fig. 2 (bottom)

Initial Collection. We collect index entries in a generic top-down traversal, which needs to be instantiated with language-specific name binding and scope rules. During the traversal, a dictionary S is maintained to keep track of the current scope for each namespace. At each node, we perform the following actions:

1. If the node is the context of a binding instance of name n in namespace ns, we create a new unique qualifier q, construct URIs $u' = S(ns)/ns.n$ and $u = u'.q$, and add (u', u) to B. If the instance is of type t, we add (u, t) to P_{type}. If the node is a scope for a namespace ns', we update $S(ns)$ to u.
2. If the current node is an anonymous scope for a namespace ns, we extend $S(ns)$ with an additional anonymous segment.
3. If the current node defines an alias, transitive, or non-transitive wildcard import, we add corresponding pairs of URIs to A, $TI_{\texttt{ns}}$, or $NI_{\texttt{ns}}$.

Collection does not consider binding prospects which need to be resolved. Furthermore, entries in $TI_{\texttt{ns}}$, $NI_{\texttt{ns}}$, and P_{type} might still require project-wide name resolution and type analysis. Instead of performing this analysis during the collection, we defer the remaining analysis tasks to a second phase of analysis and store unique placeholder URIs in the index. For example, the type of field `A.b` contains a class name `B`, which needs to be resolved. The index in Fig. 3 does not contain an actual type, but a reference to a deferred resolution task. Also, the index entries for wildcard imports refers to a deferred task, since the name of the base class of class `C` needs to be resolved first.

The semantic index is a project-wide data structure, but collection can be split over separate partitions. A *partition* is typically a file, but can also be a smaller unit. The only constraint we impose on partitions is that they need to be in global scope. This ensures that index collection is independent of other partitions. Collection for a partition p will provide us with a partial index consisting of B_p, A_p, $TI_{p,ns}$, $NI_{p,ns}$, and $P_{p,type}$. The overall index can be formed by combining all partial indices of a project.

Incremental Collection. When a partition is edited, reanalysis is triggered. But only the partial index of the changed partition needs to be recollected, while partial indices of other partitions remain valid. Partial recollection will result in an updated relation B'_p. Given the original B_p, we define a change set $\Delta_B = (B'_p \setminus B_p) \cup (B_p \setminus B'_p)$ of entries added to or removed from B. In the same way, we can define Δ_A, and $\Delta_{P_{type}}$. For imports, the situation is slightly different, since we need to consider changes in transitive import chains. We keep a change set $\Delta_{I_{ns}}$ for a derived relation $I_{ns} = TI^*_{ns} \circ NI_{ns}$, where TI^* is the reflexive transitive closure of TI and I is the composition of this closure with NI.

Example. Fig. 3 shows non-empty change sets for the running example. Thereby, superscripts indicate editing steps. In step 1, changing the return type of method `A.m` causes a change in P_{type}. In step 3, adding a parameter to the same method causes changes to B and P_{type}. In step 4, changing the parent of class `C` causes changes in I_{Field} and I_{Method}. In step 5, enclosing class `C` in a namespace affects all index entries for the class and its contained elements. The next section discusses how change-sets trigger reevaluation of deferred analysis tasks.

4 Deferred Analysis Tasks

In the previous section, we discussed the collection of index entries. This collection is efficient, since it requires only a single top-down traversal. When a partition changes, recollection is even more efficient, since it can be restricted to the changed partition, while the collected entries from other partitions remain valid. This is achieved by deferring name resolution and type analysis tasks, which might require information from other partitions or from other tasks.

Tasks are collected together with index entries and evaluated afterwards in a second analysis phase. For evaluation, no traversal is needed. Instead, inter-task dependencies determine an evaluation order. When a partition changes, only the tasks for this partition are recollected in the first phase. Change sets determine which tasks need to re-evaluated, including affected tasks from other partitions.

Instructions. Each *task* consists of a special URI, which is used as a placeholder in the semantic index, its dependencies to other tasks, and an instruction. Fig. 4 lists the instructions which can be used in tasks. Their semantics is given with respect to the semantic index, a type cast relation $C \subseteq \mathcal{T} \times \mathcal{T}$, where $(t, t') \in C$ iff type t can be cast to type t', and a partial function $\delta_C : \mathcal{T} \times \mathcal{T} \to \mathbb{N}$ for the distance between types. We write $R\,[S]$ to denote the image of a set S under a

Instruction	**Semantics**
resolve uri	$B\,[\texttt{uri}]$
resolve alias uri	$A\,[\texttt{uri}]$
resolve import ns **into** uri	$I_{\texttt{ns}}\,[\texttt{uri}]$
lookup type of uri	$P_{type}\,[\texttt{uri}]$
check type t **in** T	$\{\texttt{t}\} \cap \texttt{T}$
cast type t **to** T	$C\,[\texttt{t}] \cap \texttt{T}$
assign type t	$\{\texttt{t}\}$
s1 + s2	$R\,[\texttt{s1}, \texttt{s2}]$
s1 <+ s2	$\begin{cases} R\,[\texttt{s1}]\text{ , if } \neq \emptyset \\ R\,[\texttt{s2}]\text{ , otherwise} \end{cases}$
filter s1 + s2 **by type** T	$\{\, u \in R\,[\texttt{s1}, \texttt{s2}] \mid P_{type} \circ C\,[u] \cap \texttt{T} \neq \emptyset \}$
filter s1 <+ s2 **by type** T	$\begin{cases} \{\, u \in R\,[\texttt{s1}] \mid (P_{type} \circ C)\,[u] \cap \texttt{T} \neq \emptyset \}\text{ , if } \neq \emptyset \\ \{\, u \in R\,[\texttt{s2}] \mid P_{type} \circ C\,[u] \cap \texttt{T} \neq \emptyset \}\quad \text{, otherwise} \end{cases}$
disambiguate s1 + s2 **by type** T	$\{\, u \in R\,[\texttt{s1}, \texttt{s2}] \mid \forall u' \in R\,[\texttt{s1}, \texttt{s2}] : \delta_C(u', \texttt{T}) \geq \delta_C(u, \texttt{T}) \}$
disambiguate s1 <+ s2 **by type** T	$\begin{cases} \{\, u \in R\,[\texttt{s1}] \mid \forall u' \in R\,[\texttt{s1}, \texttt{s2}] : \delta_C(u', \texttt{T}) \geq \delta_C(u, \texttt{T}) \}\text{ , if } \neq \emptyset \\ \{\, u \in R\,[\texttt{s2}] \mid \forall u' \in R\,[\texttt{s1}, \texttt{s2}] : \delta_C(u', \texttt{T}) \geq \delta_C(u, \texttt{T}) \}\text{ , ow.} \end{cases}$

Fig. 4. Syntax and semantics of name and type analysis instructions. `uri` denotes a URI, `ns` a namespace, `t` a type, `T` a set of types, and `s1`, `s2` subtask IDs.

relation R and omit set braces for finite sets, that is, we write $R[e]$ instead of $R[\{e\}]$. We provide three name resolution instructions for looking up binding instances from B (**resolve**), named imports from A (**resolve alias**), and wildcard imports from the derived relation I_{ns} (**resolve import**), and four type analysis instructions for type look-up from P_{type} (**lookup**), for checks with respect to expected types (**check**), for casts to an expected type according to C (**cast**), and for assigning types to program elements (**assign**).

Example. Fig. 5 shows tasks and their solutions for the running example. Tasks `1` to `6` try to resolve class name `B`. Task `1` looks for `B` directly in the global scope. It finds an entry in B and *succeeds*. Task `2` looks for aliases, which task `3` tries to resolve next. Instead of a concrete URI, the task `3` has a reference to task `2`. Since task `2` *fails* to find any named imports, task `3` also fails. Task `5` tries to resolve `B` inside imported scopes, which are yielded by task `4`. Both tasks fail. Task `6` combines resolution results based on local classes, aliases, and imported classes. We will discuss such combinators in the next example.

Tasks `7` to `25` are involved in type checking the return expression of `A.m()` in Fig. 1. Task `7` assigns type **int** to the integer constant. Tasks `8` to `18` are an example for the interaction between name and type analysis. The first six tasks try to resolve `b` either as a local variable, a field in the current class, or an inherited field. Next, task `14` looks up the type of the resolved field `A.b`, before the remaining tasks resolve field `f` with respect to that type `B`. Task `19` looks up the type of the referred field. The remaining tasks analyse the binary expression: Tasks `20` and `21` check if the subexpressions are numeric or string types. Tasks `22` and `23` try to coerce the left to the right type and vice versa. Both tasks are combined by task `24`. Finally, task `25` checks if the type of the return expression can be coerced to the declared return type of the method.

Combinators. Fig. 4 also shows six instructions to combine the results of subtasks. The semantics of these combinators are expressed in terms of a relation R, where $(t, r) \in R$ iff r is a result of task t. Notably, tasks can have multiple results. We will revisit R later, when we discuss task evaluation.

The simplest combinators are a non-deterministic choice `+` and a deterministic pendant `<+`. The result of the non-deterministic choice is the union of the results of its subtasks. while the result of the deterministic choice is the result of its first non-failing subtask. Furthermore, we provide combinators **filter** and **disambiguate**. Both can be used in a non-deterministic or deterministic fashion to combine the result sets of resolution tasks with respect to expected types. **filter** keeps only compliant results. **disambiguate** keeps only results which fit best with respect to the expected types. The non-deterministic variant keeps all of them, while the deterministic variant chooses the first subtask which contributes to the best fitting results.

Example. In Fig. 5, task `6` combines resolution results based on local classes, aliased classes, and imported classes. The non-deterministic choice ensures that no result is preferred over another. Similarly, task `24` combines the results of alternative coercion tasks. In tasks `12` and `13`, deterministic choices ensure that local fields win over inherited fields and variables win over fields, respectively.

ID	Instruction	Results
1	**resolve** C#:/*Class*.B	C#:/*Class*.B.1
2	**resolve alias** C#:/*Class*.B	
3	**resolve** Task:/2	
4	**resolve import** *Class* **into** C#:/	
5	**resolve** Task:/4/*Class*.B	
6	Task:/1 + Task:/3 + Task:/5	C#:/*Class*.B.1
7	**assign type** int	int
8	**resolve** C#:/*Class*.A.1/*Method*.m.1/*Var*.b	
9	**resolve** C#:/*Class*.A.1/*Field*.b	C#:/*Class*.A.1/*Field*.b.1
10	**resolve import** *Field* **into** C#:/*Class*.A.1	
11	**resolve** Task:/10/*Field*.b	
12	Task:/9 <+ Task:/11	C#:/*Class*.A.1/*Field*.b.1
13	Task:/8 <+ Task:/12	C#:/*Class*.A.1/*Field*.b.1
14	**lookup type of** Task:/13	C#:/*Class*.B.1
15	**resolve** Task:/14/*Field*.f	C#:/*Class*.B.1/*Field*.f.1
16	**resolve import** *Field* **into** Task:/14	
17	**resolve** Task:/16/*Field*.f	
18	Task:/15 <+ Task:/17	C#:/*Class*.B.1/*Field*.f.1
19	**lookup type of** Task:/18	float
20	**check type** Task:/7 **in** {int, long, float, double, String}	int
21	**check type** Task:/19 **in** {int, long, float, double, String}	float
22	**cast type** Task:/21 **to** Task:/20	
23	**cast type** Task:/20 **to** Task:/21	float
24	Task:/22 + Task:/23	float
25	**cast type** Task:/24 **to** float	float
26	**cast type** Task:/20 **to** int	int
27	**resolve** C#:/*Class*.A	C#:/*Class*.A.1
28	**resolve alias** C#:/*Class*.A	
29	**resolve** Task:/28	
30	**resolve** Task:/4/*Class*.A	
31	Task:/27 + Task:/29 + Task:/30	C#:/*Class*.A.1
32	**resolve** C#:/*Class*.C.1/*Method*.m	
33	**resolve import** *Method* **into** C#:/*Class*.C.1	C#:/*Class*.A.1
34	**resolve** Task:/33/*Method*.m	C#:/*Class*.A.1/*Method*.m.1
35	**assign type** []	[]
36	**disambiguate** Task:/32 <+ Task:/34 **by type** Task:/35	C#:/*Class*.A.1/*Method*.m.1
37	**lookup type of** Task:/36	([], float)
38	**cast type** Task:/37 **to** int	

Fig. 5. Tasks and their solutions for the C# program in Fig. 1

Method call resolution in the presence of overloaded methods is a well-known example for interaction between name and type analysis. Actual and formal argument types need to be considered by the resolution, since they need to comply. Furthermore, relations between these types indicate which declaration is more applicable. As an example, consider tasks `32` to `36` in Fig. 5. They resolve method call `m()` in the return expression of `C.n()` from Fig. 1. Task `32` tries to resolve it locally, while tasks `33` and `34` consider inherited methods. Task `35` assigns an empty list as the type of the actual parameters of the call. Task `36` selects only these methods which fits this type best, preferring local over inherited methods. Finally, the last two tasks check the return expression of `C.n`. Task `37` looks up the type of `A.m`. Task `38` tries to casts this to the declared return type, but fails.

Initial Evaluation. During the generic traversal in the collection phase, we do not only collect semantic index entries but also instructions of tasks ($T \subseteq \mathcal{U} \times \mathcal{I}$) and inter-task dependencies ($D \subseteq \mathcal{U} \times \mathcal{U}$). Language-specific collection rules are needed to control the collection of name resolution and type analysis tasks. D imposes an evaluation order for tasks. First, we can evaluate independent tasks. Next, we can evaluate tasks which only depend on already evaluated tasks. This will evaluate all tasks except those with cyclic dependencies, which we consider erroneous. As mentioned earlier, we capture task results in a relation $R \subseteq \mathcal{U} \times (\mathcal{U} \cup \mathcal{T})$.

The instruction of each task is evaluated according to the semantics given in Fig. 4. However, this only works, if we replace placeholders of dependent subtasks with their results. When a subtask has multiple results, we evaluate the dependent task for each of these results. Consider task `14` from Fig. 5 as an example. It can only be evaluated after replacing the placeholder `Task:/13` with a result of the corresponding task. Since this task has a single result `C#:/Class.A.1/Field.b.1`, we actually need to evaluate the instruction **`lookup type`** `C#:/Class.A.1/Field.b.1`, yielding `C#://Class.B.1` as its only result.

Incremental Evaluation. When a partition is edited, the partial index and tasks for this partition will be recollected, resulting in an updated relation T'_p. We need to evaluate new tasks, which did not exist in another partition before. We collect the URIs of these tasks in a change set: $\Delta_{T_p} = \mathrm{dom}(T'_p \setminus T_p)$. Furthermore, a changed semantic index might affect the results of the tasks from all partitions, requiring the reevaluation of those tasks. The various change sets determine which tasks need to be reevaluated:

$(u', u) \in \Delta_B$**:** tasks which evaluated an instruction **`resolve`** u'.
$(a, u) \in \Delta_A$**:** tasks which evaluated an instruction **`resolve alias`** a.
$(u', u) \in \Delta_I$**:** tasks which evaluated an instruction **`resolve import`** u'.
$(u, t) \in \Delta_{P_{type}}$**:** tasks which evaluated an instruction **`lookup type of`** u and **`filter`** or **`disambiguate`** tasks with a subtask s with $u \in R[s]$.

We maintain the URIs of these tasks in another change set Δ_T. The URIs of tasks which require evaluation is given by the set $\Delta_{T_p} \cup D^*[\Delta_T]$.

Example. In step 1 of the running example, task 25 becomes obsolete, since the return expression needs to be checked with respect to a new type, which is done by a new task 39, shown in Fig. 6. Furthermore, the disambiguation in task 36 depends on an element in $\Delta^1_{P_{type}}$, which is to be reevaluated. Transitive dependencies trigger also the reevaluation of tasks 37 and 38. Since task 38 succeeds now, it does no longer indicate a type error in `C.n`. But the new task 39 fails, indicating a new type error in `A.m`. In step 2, tasks 15, 17 to 19, 21 to 24, and 39 become obsolete, since another field needs to be resolved. The semantic index was not changed, and only the corresponding new tasks 40 to 48 need to be evaluated. In step 3, the additional variable parameter causes changes in the semantic index. Δ^3_B requires the reevaluation of task 8 and its dependent tasks 14, 16, and 40 to 48. Furthermore, $\Delta^3_{P_{type}}$ requires the reevaluation of task 36 and its dependent tasks 37 and 38. Similarly, $\Delta^4_{I_{Field}}$ requires the reevaluation of task 33 and its dependent tasks 34 and 36 to 38. Finally, the new enclosing namespace introduced in step 5 makes tasks 32 to 34 and 36 to 38 obsolete and introduces new tasks 49 to 61, which take the new namespace into account.

ID	Instruction	Results
39	**cast type** Task:/24 **to** int	
40	**resolve** Task:/14/*Field*.i	C#:/*Class*.B.1/*Field*.i.1
41	**resolve** Task:/16/*Field*.i	
42	Task:/40 <+ Task:/41	C#:/*Class*.B.1/*Field*.i.1
43	**lookup type of** Task:/42	int
44	**check type** Task:/43 **in** {int, long, float, double, String}	int
45	**cast type** Task:/44 **to** Task:/20	int
46	**cast type** Task:/20 **to** Task:/44	int
47	Task:/45 + Task:/46	int
48	**cast type** Task:/47 **to** int	int
49	**resolve** C#:/Ns.N.1/*Class*.B	
50	**resolve alias** C#:/Ns.N.1/*Class*.B	
51	**resolve** Task:/50	
52	**resolve import** *Class* **into** C#:/Ns.N.1	
53	**resolve** Task:/52/*Class*.B	
54	Task:/49 + Task:/51 + Task:/53	
55	Task:/31 + Task:/54	C#/*Class*.B.1
56	**resolve** C#:/Ns.N.1/*Class*.C.1/*Method*.m	
57	**resolve import** *Method* **into** C#:/Ns.N.1/*Class*.C.1	C#:/*Class*.B.1
58	**resolve** Task:/57/*Method*.m	C#:/*Class*.B.1/*Method*.m.1
59	**disambiguate** Task:/56 + Task:/58 **by type** Task:/35	C#:/*Class*.B.1/*Method*.m.1
60	**lookup type of** Task:/59	([], int)
61	**cast type** Task:/60 **to** int	int

Fig. 6. New tasks and their solutions for the C# program in Fig. 2

5 Implementation

We have implemented the approach as three components of the Spoofax language workbench [14]. The first component is a Java implementation of the semantic index. It maintains a multimap storing relations B, A, I, and P, a set keeping partition names, and another multimap from partitions to their index entries. During collection, it calculates change sets on the fly, maintaining two multisets for newly added and removed elements.

The second component is a task engine implemented in Java. It maintains a map from task IDs to their instructions and bidirectional multimaps between task IDs and their partitions, between task IDs and index entries they depend on, and for task dependencies. Just as the semantic index, the task engine exposes a collection API and calculates change sets on the fly, maintaining a set of added and a set of removed tasks. Additionally, it exposes an API for task evaluation. During evaluation, it maintains a queue of scheduled tasks and a bidirectional multimap of task dependencies which are discovered dynamically. Results and messages of tasks are kept in maps. Both components use hash-based data structures which can be persisted to file. They support Java representations of terms as values and expose their APIs to Stratego [2], Spoofax' term rewriting language for analysis, transformation, and code generation.

```
Class(NonPartial(), c, _, _): defines Class c scopes Field, Method
Field(_, f)                  : defines Field f
Method(_, m, _, _)           : defines Method m scopes Var

Base(c):
  imports Field, imported Field, Method, imported Method from Class c

ClassType(c)   : refers to Class c
FieldAcc(e, f) : refers to Field f in Class c where e has type c
VarRef(x)      : refers to Var x otherwise refers to Field x
ThisCall(m, p*): refers to best Method m of type t* where p* has type t*
```

```
overlays
  NUMERIC() = [Int(), Long(), Float(), Double()]
  STRING()  = ClassType(PackRef("System"), "String")

type-of(|ctx):
  Add(e1, e2) → <choose(|ctx)> [ty1', ty2']
  where
    ty1  := <type-check(|ctx)> (e1, [STRING() | NUMERIC()])
  ; ty2  := <type-check(|ctx)> (e2, [STRING() | NUMERIC()])
  ; ty1' := <type-match(|ctx, Coerce())> (ty1, ty2)
  ; ty2' := <type-match(|ctx, Coerce())> (ty2, ty1)
```

Fig. 7. Declarative name binding and scope rules for C# in NaBL (top) and manually written Stratego rule for typing additions and string concatenations in C# (bottom)

The third component implements index and task collection as a generic traversal in Stratego. At each tree node, the traversal applies language-specific rewrite rules for name and type analysis. These rules can either be generated from name binding and scope rules defined in NaBL, or manually written in Stratego. For example, Fig. 7 shows an extract of NaBL rules as well as a manually written Stratego rule for C#. The latter involves callbacks to the collection component, which creates the corresponding tasks in the task engine. `type-check` creates a **`check`** task, `type-match` creates a **`cast`** task, and `choose` creates a nondeterministic choice. The rule looks very similar to an ordinary typing rule in Stratego, but instead of calculating types, it calculates tasks, which are evaluated later. The API hides the internals of our approach from the language engineer, who can specify an incremental static analysis in NaBL and Stratego in the same way as a regular static analysis.

6 Evaluation

We evaluate the *correctness*, *performance*, and *scalability* of our approach with an implementation for name and type analysis of WebDSL programs. Correctness is interesting since we only analyze affected program elements. We expect incremental analysis to yield the same result as a full analysis. Performance and scalability are crucial since they are the main purpose of incremental analysis. We want to assess whether performance is acceptable for practical use in IDEs and how the approach scales for large projects. Specifically, we evaluate the following research questions: *RQ1*) Does incremental name and type analysis of WebDSL applications yield the same results as full analysis? *RQ2*) What is the performance gain of incremental name and type analysis of WebDSL applications compared to full analysis? *RQ3*) How does the size of a WebDSL application influence the performance of incremental name and type analysis? *RQ4*) Is incremental name and type analysis suitable for a WebDSL IDE?

Research Method. In a controlled setting, we quantitatively compare the results and performance of incremental and full analysis of different versions of WebDSL applications. We have reimplemented name and type analysis for WebDSL, using NaBL to specify name binding and scope rules and Stratego to specify type analysis. We apply the same algorithm to perform full and incremental analyses to the source code histories of two WebDSL applications. We run a fullanalysis on all files in a revision, and and incremental analysis only on changed files with respect to the result of a full analysis of the previous revision.

Subjects. WebDSL is a domain-specific language for the implementation of dynamic web applications [7]. It was designed from the ground up with static analysis and cross-aspect consistency checking in mind [11]. This focus makes it is an ideal candidate to evaluate its static analysis. WebDSL provides many language constructs on which constraints have to be checked. It also embodies a complex expression language that is representative of expressions in general purpose languages such as Java and C#. It has been used for several applications

in production, including the issue tracker Yellowgrass[1], which is a subject of this evaluation, the digital library Researchr, and the online education platform WebLab. When developing such larger applications, the usability of the WebDSL IDE sometimes suffered from the lack of incremental analyses. We focus on two open source WebDSL applications, Blog, a web application for wikis and blogs, and Yellowgrass, a tag-based issue tracker. In their latest revisions, their code bases consist of approximately 7 and 9 KLOC.

Data collection. We perform measurements by repeating the following for every revision of each application. We run an incremental and a full analysis. During each of the analyses we record execution timings. After each analysis we preserve the data from the semantic index and the task engine which we analyse afterwards. Each analysis is sequentially executed on command line in a separate invocation of the Java Virtual Machine (JVM) and garbage collection is invoked before each analysis. After starting the virtual machine, we run three analyses and discard results allowing for the warmup period of the JVM's JIT compiler. All executions are carried out on the same machine with 2.7 Ghz Intel Core i-7, 16 GB of memory, and Oracle Java Hotspot VM version 1.6.0 45 in server JIT mode. We fix the JVM's heap size at 4 GB to decrease the noise caused by garbage collection. We set the maximum stack size at 16 MB.

Analysis procedure. For *RQ1*, we evaluate the structural equality of data from the semantic index and the task engine produced by full and incremental analysis. For *RQ2*, we determine absolute execution times of full and incremental analysis and the relative speed up. We calculate the relative performance gain between analyses separately for each revision. We report geometric mean and distribution of absolute and relative performance of all revisions. For *RQ3*, we determine the number of lines and the number of changed lines of a revision. We relate the incremental analysis time to these numbers. For *RQ4*, we filter revisions which changed only a single file. On these revisions, we determine the execution time of incremental analysis.

Results and Interpretation. We published the collected data and all analysis results in a public repository[2], including instructions on reproducing our experiments. Since both applications yield similar results, we discuss only Yellowgrass data here. Data for Blog can be found in the repository. For the future, we plan to collect data on more WebDSL applications and on more programming languages. Our implementation and the subjects are also open source.

RQ1) For all revisions of both applications, incremental and full analysis produce structurally equal data in semantic index and task engine. This is the expected outcome and supports the equivalence of both analyses.

RQ2) Fig. 8 show the absolute execution times of full and incremental analyses of all revisions. Full analysis takes between 4.74 and 13.31 seconds. Incremental analysis takes between 0.37 and 4.97 seconds. The mean analysis times are 9.75 seconds and 0.96 seconds, with standard deviations of 2.29 and 0.61

[1] `http://yellowgrass.org`
[2] `https://bitbucket.org/slde/opendata-experiments`

seconds, respectively. Incremental analysis takes between 3.06% and 43.75% of the time of a full analysis. The mean ratio between incremental and full analysis is 10.56%. Thus, incremental analysis gives huge performance gains.

RQ3) Fig. 9 shows incremental analysis times per revision, ordered by LOC and changed LOC, respectively. The size of a project does not seem to influence incremental analysis time (correlation coefficient −0.18), but the size of the change does. This is the expected outcome, but more experiments will be needed.

RQ4) There were 137 revisions which affected only a single file. Incremental analysis takes between 0.37 and 1.12 seconds. There is only one revision where incremental analysis takes longer than one second. The mean incremental analysis time is 0.56 seconds. All analysis times would be acceptable response times in an interactive IDE setting, where analysis is performed in the background without blocking the user interface. Single responses which take slightly more than one second would still be acceptable, if regular responses are fast. Furthermore,

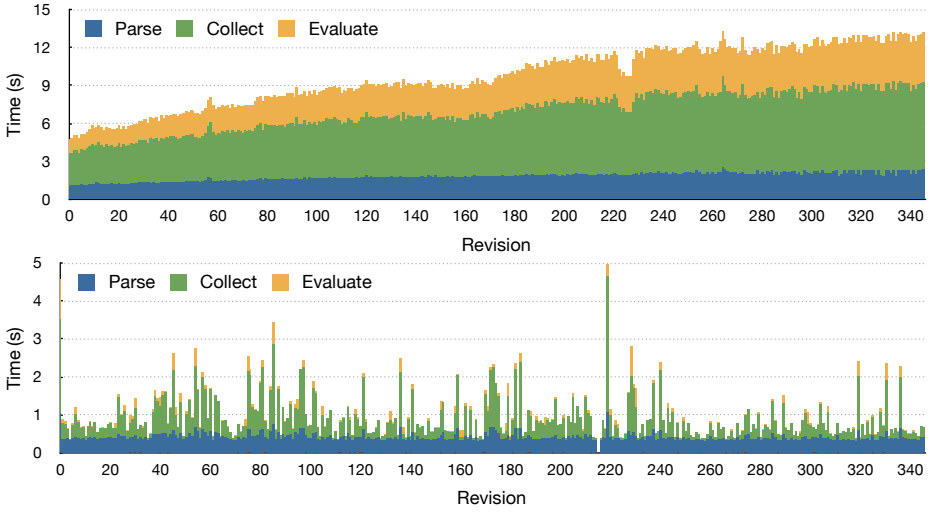

Fig. 8. Analysis time for full (top) and incremental (bottom) analyses

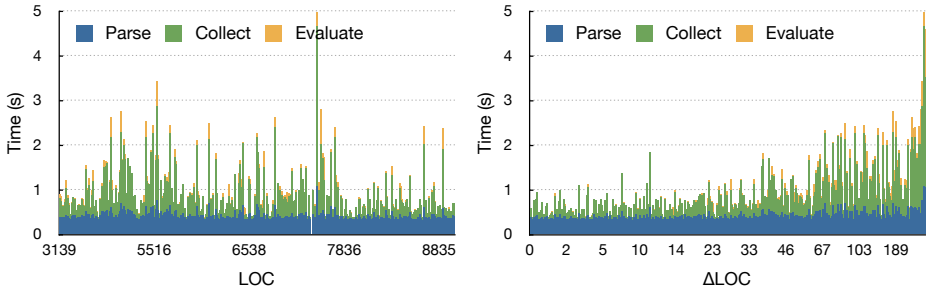

Fig. 9. Incremental analysis time ordered by LOC (left) and ΔLOC (right)

changes between two revisions are more coarse grained and should require more re-evaluation than changes in an editing scenario.

Threats to Validity. An important threat to external validity is that we analyzed only WebDSL applications and only two of them. We are convinced that WebDSL's name and type analysis is representative for other languages, but our evaluation cannot generalize beyond WebDSL and its sublanguages. Furthermore, other WebDSL applications, particularly those of different size, might show different characteristics. Additional threats are the large distance between revisions and the correctness of revisions. In real-time editing scenarios, distances might be much smaller and revisions might switch between correct and erroneous states. We believe that smaller distances would only be in the benefit of incremental analysis. Erroneous revisions should not affect parse and collection times but evaluation times, which tend to be small. A threat to internal validity is file size. Incremental analysis re-parses and re-collects changed files. Independent of the actual changes inside a file, file size alone can influence parse and collection times. However, we believe that this does not influence the conclusions from any of our research questions. Regarding construct validity, we measured performance using wall-clock time only and control JIT compilation with a warm-up phase. By running the garbage collector between analysis runs, we ensured a similar amount of memory available to all analyses. However, the semantic index and the task engine store large amounts of data (13 MB in the worst case) and may experience garbage collection pauses.

7 Related Work

We give an overview of other approaches for incremental name and type analysis.

IDEs and Language Workbenches. IDEs such as Eclipse typically lack a generic framework for the development of incremental analyses, but provide manual implementations of incremental analysis and compilation for popular languages such as Java or C#. Some language workbenches automatically derive incremental analyses. In SugarJ [5], extensions inherit the incremental behaviour of SugarJ, which uses the module system of Java to provide incremental compilation on file-level, but lacks name and type analysis of its host language Java. Xtext [6] leverages incremental analysis and compilation from the Eclipse JDT to user-defined languages, as long as they map to Java concepts. The JDT performs only local analyses on edit and global analyses on save. MPS [30] does not require name binding due to its projectional nature. It supports incremental type analysis but lacks a framework for other incremental analyses. In general, language workbenches lack frameworks for developing incremental analyses.

Attribute Grammars. Attribute grammars [15] provide a formal way of specifying the semantics of a context-free language, including name and type analysis. One of the first incremental attribute evaluators is proposed in [3]. It only evaluates changed attributes and propagates evaluation to affected attributes. A similar incremental evaluation algorithm is shown in [31,32] for ordered attributed grammars [13]. In [22,24,23,12], extensions to propagation are shown that stop propagation if an attribute value is unchanged from its previous attribution.

Similar to attribute grammars, our approach exploits static dependencies, caching, and change propagation. Similar to ordered attribute grammars, we assume an evaluation order of tasks. Though tasks can be cyclic, we just do not evaluate them. While attributes are (re-)evaluated in visits to the tree, our collection separates tasks from the tree and they are (re-)evaluated independent of the tree. As a consequence, we do not require incremental parsing techniques and are not restricted to editing modes. For name analysis, attribute grammars typically pass environments throughout the tree. Incremental name analysis suffers from this as a single change in the environment requires a full re-evaluation of the aggregated environment and all dependent attributes. In our approach, we have a predefined notion of an environment, the semantic index, which is globally maintained. It enables fine-grained dependency tracking for name and type analysis tasks solely based on changing entries, not on changing environments.

Reference Attribute Grammars. A popular extension to attribute grammars is the addition of reference attributes. These simplify the specification of algorithms that require non-local information, including name resolution. Door Attribute Grammars [8,9] extend attribute grammars with reference attributes and door objects which facilitate analysis of object-oriented languages. A similar but more general extension is shown in [21]. Reference Attributed Grammars [10] are a generalization of door attribute grammars where the door objects are removed. In [26], an incremental evaluator for reference attributed grammars is shown which is used by the JastAdd [4] meta-compilation system. JastAdd also adds parametrized attributes which allow attributes to be parametrized, forming a mapping. The approach is compared to traditional attribute grammars in [27] and shows that the use of reference attribute grammars reduces the number of affected attributes for name and type analysis significantly.

Our approach has two mechanisms similar to reference attributes. First, we can refer to binding instances by URIs and can look up their properties in the semantic index. Second, properties and tasks can refer to arbitrary other tasks. Reference attribute grammars discover dependencies during evaluation. We detect inter-task dependencies after collection. This already helps in establishing an ordering for evaluation. Only dependencies from properties to tasks are discovered during evaluation. Similar to ordinary attribute grammars, reference attribute grammars also do not provide a solution for aggregate attributes.

Some attribute grammar formalisms take a functional approach to evaluation. In [19] attributes are evaluated using visit-functions with memoization. A more general extension to attribute grammars is the higher order attribute grammar [28,25] for which an incremental evaluator is presented in [29]. Similar

to this approach, our approach employs a global cache and uses hash consing to efficiently share tasks and to make look-ups into the cache extremely fast. Tasks can also be seen as functions, but the evaluation strategy differs. Visit-functions are still applied on subtrees while tasks are completely separated from the tree.

Other Approaches. Pregmatic [1] is an incremental program environment generator that uses extended affix grammars for specification. It uses an incremental propagation algorithm similar to the one used by attribute grammar approaches which were discussed earlier. Instead of separating parsing and semantic analysis, all evaluation is done during parse-time which differs significantly from our parse, collect and evaluate approach. Incremental Rewriting [18] describes efficient algorithms for incrementally rewriting programs based on algebraic specifications. An algorithm for incrementally evaluating functions on aggregated values is also shown. The approach does not support non-local dependencies, making specification of name binding less intuitive as it requires copying of information.

8 Conclusion

We have proposed an approach for incremental name and type analysis in two phases, collection and deferred evaluation of analysis tasks. The collection is instantiated with language-specific name binding and type rules and incremental on file level. Unchanged files are neither re-parsed nor re-traversed. The evaluation phase is incremental on task level. When a file changes, all tasks that are affected by this change are reevaluated. This might include dependent tasks from other files.

Tasks execute low-level instructions for name resolution and type analysis, and can form a basis for the definition of declarative meta-languages at a higher level of abstraction. For example, we map declarative name binding and scope rules expressed in NaBL to an instantiation of the presented approach. We implemented the approach as part of the Spoofax language workbench. It frees language engineers from the burden of manually implementing incremental analysis. We applied the implementation to WebDSL and empirical evaluation has shown this analysis to be responsive to changes in analyzed programs and suitable to the interactive requirements of an IDE setting.

Acknowledgements. This research was supported by NWO/EW Free Competition Project 612.001.114 (Deep Integration of Domain-Specific Languages) and by a research grant from Oracle Labs. We would like to thank Lennart Kats for his contribution to the start of NaBL and to Spoofax' incremental analysis project. We would also like to thank Karl Kalleberg for valuable discussions on the interpretation of name binding and scoping rules.

References

1. van den Brand, M.G.J.: PREGMATIC - a generator for incremental programming environments. Ph.D. thesis, University Nijmegen (1992)
2. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. SCP 72(1-2), 52–70 (2008)
3. Demers, A.J., Reps, T.W., Teitelbaum, T.: Incremental evaluation for attribute grammars with application to syntax-directed editors. In: POPL, pp. 105–116 (1981)
4. Ekman, T., Hedin, G.: The jastadd system - modular extensible compiler construction. SCP 69(1-3), 14–26 (2007)
5. Erdweg, S., Rendel, T., Kástner, C., Ostermann, K.: Sugarj: Library-based syntactic language extensibility. In: OOPSLA, pp. 391–406 (2011)
6. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: OOPSLA, pp. 307–309 (2010)
7. Groenewegen, D.M., Hemel, Z., Kats, L.C.L., Visser, E.: WebDSL: a domain-specific language for dynamic web applications. In: OOPSLA, pp. 779–780 (2008)
8. Hedin, G.: Incremental static-semantic analysis for object-oriented languages using door attribute grammars. In: SAGA, pp. 374–379 (1991)
9. Hedin, G.: Incremental Semantic Analysis. Ph.D. thesis (1992)
10. Hedin, G.: Reference attributed grammars. Informatica SI 24(3) (2000)
11. Hemel, Z., Groenewegen, D.M., Kats, L.C.L., Visser, E.: Static consistency checking of web applications with WebDSL. JSC 46(2), 150–182 (2011)
12. Johnson, G.F., Fischer, C.N.: A meta-language and system for nonlocal incremental attribute evaluation in language-based editors. In: POPL, pp. 141–151 (1985)
13. Kastens, U.: Ordered attributed grammars. ACTA 13, 229–256 (1980)
14. Kats, L.C.L., Visser, E.: The Spoofax language workbench: rules for declarative specification of languages and IDEs. In: OOPSLA, pp. 444–463 (2010)
15. Knuth, D.E.: Semantics of context-free languages. MST 2(2), 127–145 (1968)
16. Konat, G., Kats, L., Wachsmuth, G., Visser, E.: Declarative name binding and scope rules. In: Czarnecki, K., Hedin, G. (eds.) SLE 2012. LNCS, vol. 7745, pp. 311–331. Springer, Heidelberg (2013)
17. Krishnamurthi, S.: Programming Languages: Application and Interpretation (2007)
18. Meulen, E.A.V.D.: Incremental Rewriting. Ph.D. thesis, University of Amsterdam (1994)
19. Pennings, M.C.: Generating incremental attribute evaluators. Ph.D. thesis, Computer Science, Utrecht University (November 1994)
20. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)
21. Poetzsch-Heffter, A.: Programming language specification and prototyping using the max system. In: PLIPL, pp. 137–150 (1993)
22. Reps, T.W.: Optimal-time incremental semantic analysis for syntax-directed editors. In: POPL, pp. 169–176 (1982)
23. Reps, T.W.: Generating language-based environments. Massachusetts Institute of Technology, Cambridge (1984)
24. Reps, T.W., Teitelbaum, T., Demers, A.J.: Incremental context-dependent analysis for language-based editors. TOPLAS 5(3), 449–477 (1983)
25. Swierstra, S.D., Vogt, H.: Higher order attribute grammars. In: SAGA. pp. 256–296 (1991)

26. Söderberg, E.: Contributions to the Construction of Extensible Semantic Editors. Ph.D. thesis (2012)
27. Söderberg, E., Hedin, G.: A comparative study of incremental attribute grammar solutions to name resolution (2012)
28. Vogt, H., Swierstra, S.D., Kuiper, M.F.: Higher-order attribute grammars. In: PLDI, pp. 131–145 (1989)
29. Vogt, H., Swierstra, S.D., Kuiper, M.F.: Efficient incremental evaluation of higher order attribute grammars. In: PLIPL, pp. 231–242 (1991)
30. Völter, M., Solomatov, K.: Language modularization and composition with projectional language workbenches illustrated with MPS. In: SLE (2010)
31. Yeh, D.: On incremental evaluation of ordered attribute grammars. BIT 23(3), 308–320 (1983)
32. Yeh, D., Kastens, U.: Improvements of an incremental evaluation algorithm for ordered attribute grammars. SIGPLAN 23(12), 45–50 (1988)