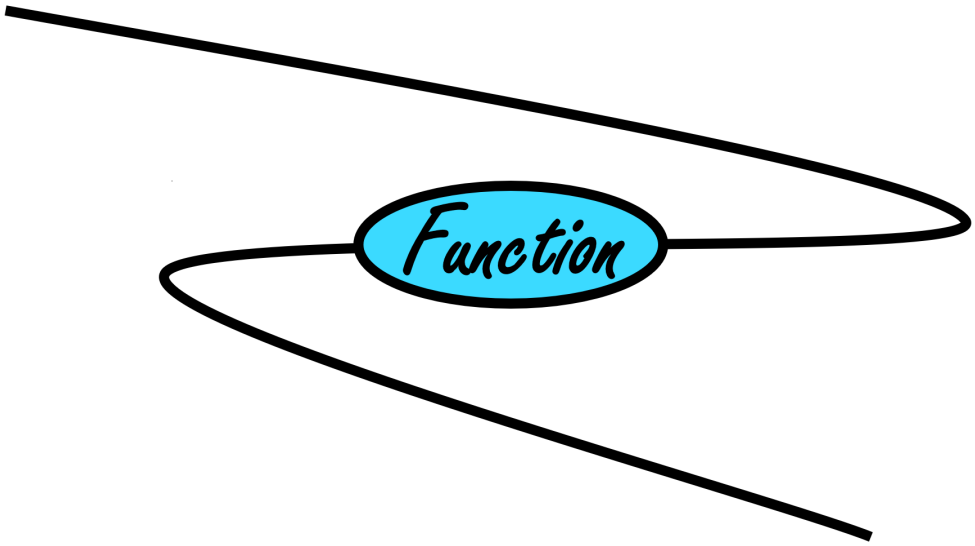
Function Inlining as a Language Parametric Refactoring

Master's Thesis



Loek van der Gugten

Function Inlining as a Language Parametric Refactoring

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Loek van der Gugten
born in Arnhem, the Netherlands

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cover picture: A function inside a line.

Function Inlining as a Language Parametric Refactoring

Author: Loek van der Gugten
Student id: 5425131
Email: lvdgugten@gmail.com

Abstract

Refactorings are program transformations that preserve the observable behavior of the program. The refactoring *function inlining* replaces a function call with the contents of the referenced function definition. To preserve the behavior, properties such as reference relations must be retained and language constructs like 'return' statements must be replaced. Implementing a behavior preserving refactoring is a time-consuming and error-prone task. In the past, such refactorings have only been implemented for one language at a time, thus they cannot be reused in other languages.

This thesis presents a language parametric function inlining algorithm that can be applied to any language. The algorithm performs function inlining and checks the transformed program for static semantics errors and changes to reference relations that would cause a change to the behavior of the program, fixing them where possible. The required language parameters are an AST parser, a static semantics analyzer that can derive reference relations, and a set of language-specific functions that can perform language-specific tasks, for example identifying return statements. An implementation of the algorithm is provided in the language workbench Spoofax using the transformation language Stratego. It can be applied to all languages with an SDF3 defined grammar and a Statix semantics analyzer. The implementation is tested on C++, WebDSL and Tiger using unit tests.

Thesis Committee:

Chair:	Prof. Dr. A. Van Deursen, Faculty EEMCS, TU Delft
Committee Member:	Prof. Dr. C.B. Poulsen, Faculty EEMCS, TU Delft
University Supervisor:	L. Miljak, Faculty EEMCS, TU Delft

Thesis Advisor:	Prof. Dr. E. Visser, Faculty EEMCS, TU Delft

Preface

Before you lies the thesis "Function Inlining as a Language Parametric Refactoring", a research project that develops the inline function refactoring in a generic manner. The thesis represents my graduation project for the Master's Degree Computer Science at the EWI faculty of TU Delft. The project ran from November 2021 to June 2022.

First and foremost I would like to offer special thanks to Prof. Eelco Visser for shaping my thesis topic and providing me with excellent guidance during the initial stages. Although no longer with us, his extensive work on the field of language development has realized and inspired many students and scientists over the course of his career.

The topic of the thesis is a follow-up on the thesis "Renaming for Everyone" by Phil Misteli, who developed a language parametric renaming refactoring as part of his Master's thesis. The research on a generic function inlining algorithm was challenging, especially because theoretically the number of possible language-specific constructs that affect function inlining is infinite. My initial goal was to cover as much as possible generically, but for some aspects this turned out to be impossible. It took me some time to accept the usage of language parameters for those aspects instead.

I would like to thank my supervisor Luka Miljak who has always been available for discussions and questions about my research. I would also like to thank Casper Poulsen for picking up Eelco's supervision role after his unfortunate passing, despite the fact that it brought a lot of other tasks on his plate at the same time.

In addition, I would like to thank Aron Zwaan and Jeff Smits for their excellent support whenever I had technical issues with Statix or Stratego in Spoofax. Finally, I would like to thank my parents Dick and Tinka for their unconditional support during my Master's Degree, especially during the Covid lockdowns.

Loek van der Gugten
Delft, the Netherlands
June 15, 2022

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A *refactoring* is a transformation of a program that preserves the observable behavior of the program. Martin Fowler, a famous figure in the world of refactoring, describes them as follows in his aptly titled book "Refactoring": "... Refactoring is a controlled technique for improving the design of an existing code base. Its essence is applying a series of small *behavior-preserving transformations* ..." [1]. Examples of types of refactorings include renaming a variable, moving a method and of course inlining a function.

With small programs, performing a single refactoring by hand is perfectly doable. Renaming a local variable should only require the transformation of a few occurrences of the variable. However, when the code base grows larger, so does the occurrence count of the variable. Not only does this increase the number of small changes you have to apply, but even more importantly it increases the risk that an unnoticed behavior change takes place, for example due to overlapping variable names. Especially with larger refactorings like inlining a function call, it can quickly become a tedious process to perform them manually. Consequently, for sufficiently large programs the automation of common refactorings has the potential to save a lot of time.

Language Parametric. The first formal mention of automated refactorings comes from Opdyke in 1992 [2]. Since then many refactorings have been implemented as part of editor services. However, refactorings are implemented ad hoc. If a refactoring is implemented for language A but you want to use it for language B, you have to re-implement the complete refactoring for language B before it can be used there. Now, developing a refactoring is useful, but also time-consuming. A wrong implementation can introduce breaking errors when applied in practice [3, 4, 5]. What if we just had to implement each refactoring once, and could immediately use it for all languages? What if, instead of multiple ad hoc implementations, we only need a single implementation?

A refactoring algorithm that works for all languages is called a *language parametric* refactoring. It is not a completely *generic* algorithm, because it will always require the relevant parts of the language's grammar and static semantics to be able to apply valid syntax transformations and ensure behavior preservation. You could consider the encoding of the language to be the language parameter and a working refactoring implementation for that language to be what is returned, hence the term language parametric refactoring. The encoding of a language is a combination of a grammar- and static semantics specification. To be able to work with the language encoding, the grammar specification has to parse programs in a language to a structured representation. The static semantics specification needs to provide API calls that return generic information such as which identifier references what declaration. Parsing and static semantic analysis happen at compile time, before runtime. So, if relevant parts of a language's semantics are dynamic (for example *dynamic name binding*) the language parametric refactoring cannot (always) be used on programs in the language.

Inline Function. In this thesis we contribute a language parametric algorithm of the *inline function refactoring*. Function inlining is the process of replacing a function call with its function definition. The parameters of the function definition are replaced by the arguments to the function call, the body is inserted above the function call and the returned value replaces the call expression. A basic example of inlining a function call can be seen in Figure 1.1 (note that generally code fragments in this thesis are in C++ unless stated differently).

```
1  int foo(int x) {
2    return x + 1;
3  }
4
5  int main() {
6    foo(2);
7  }
```

```
1  int foo(int x) {
2    return x + 1;
3  }
4
5  int main() {
6    int x = 2;
7    x + 1;
8  }
```

Figure 1.1: A very basic example of the inline function refactoring. The function call `foo(2)` in the program on the left is inlined on the right.

So, why would anyone want to inline a function call in the first place? After all, most modern compilers already apply function inlining in the compiled code whenever it benefits performance. The answer to that is as follows. A refactoring is mostly intended to improve readability of source code. Most code bases are constantly evolving and are read by multiple programmers, so it is important for the code to be as readable as possible. Function calls can improve the readability, for example if the function performs a clearly isolated task that is repeated throughout the program. However, you can also overdo splitting tasks into separate functions. When a task consists of only one or two lines, introducing a new function for the task can do more harm than good to the readability, because you have to go to a different point in the program just to read those few lines.

It can also be the case that a function call needs to be inlined as part of a bigger refactoring. For example, think about when a section of the program (including a function call) needs to be moved. If it is moved to a scope where the function definition is no longer visible, the program would break. Inlining the function call could save the change, provided that the body of the function definition works in the new scope.

Function inlining can become more complicated when the program gets more complex. Parameters might need to be initialized as new variables, name bindings of variables can change because the scope changes, duplicate definitions can occur, etc. Challenges like these are what makes an automatic inline function refactoring difficult to implement, let alone a language parametric implementation where the different static semantics of existing programming languages have to be taken into account. However, function inlining consists of a couple of major steps that apply to many languages, such as retaining reference relations and changing the parameter declarations to make them valid for insertion. Those are the steps that we cover in our algorithm, whilst using the language parameters to handle language-specific details.

Spoofax. To achieve language parametric function inlining the language workbench Spoofax [6] is used. Spoofax provides language designers a useful toolset integrated into the Eclipse IDE to aid in the development of domain specific languages (DSLs) [7, 8]. Designing a language in Spoofax can roughly be described as follows (see Figure 1.2 for an overview of the Spoofax pipeline): First, the grammar is defined declaratively using a language called Syntax Definition Formalism 3 (SDF3) [9]. This yields a parser that converts programs of the language to Abstract Syntax Trees (AST) [10] and an unparser that can do the opposite. Next, the static semantics of the language can be defined. Static semantics denotes semantics that can be checked at compile time. For many languages, this includes important concepts

such as type checking and name resolution. In Spoofax the language Statix [11, 12] is used to specify static semantics for the DSL. Once implemented, Statix generates a type checker for the target language and can perform name binding analysis.

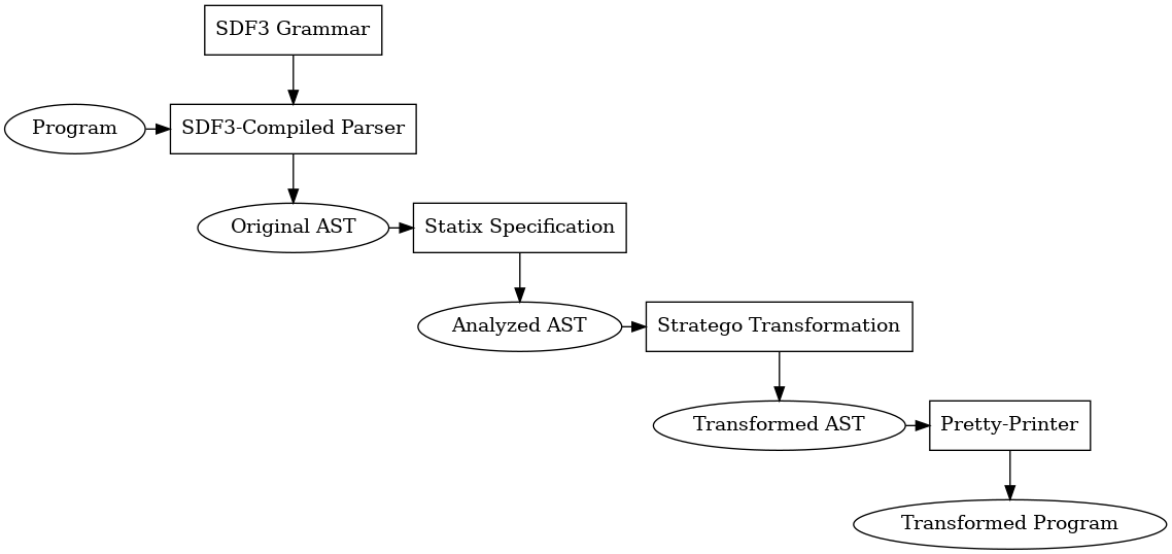


Figure 1.2: The pipeline of applying a Spoofax Stratego transformation to a program.

Once the syntax and static semantics have been defined, you can start specifying automatic transformations that make life easier for programming in practice. One way to transform a program would be to directly edit the source code, but transforming the terms in the AST representing the program is a much more structured approach. In Spoofax, transforming the AST can be performed using the Stratego language [13]. It provides built-in functions (usually referred to as strategies) that ease the traversal of the AST. Additionally, it enables interaction with the other tools in the form of API calls. For example, Stratego can query the Statix analysis with an identifier term to find the referenced declaration.

For a language parametric refactoring it makes sense to implement it using Spoofax. This way every language that is defined in Spoofax automatically has access to the refactoring. On top of that, because of the integration with the Eclipse IDE, Spoofax can generate editor plugins from language definitions that make the use of the refactoring user-friendly in practice.

Testing a language implementation and defined transformations can be performed using the SPoofax Testing language (SPT) [14]. With it, you can specify a program, a function (such as a transformation) to apply to the program and an expected result. The framework will then check if the result of applying the function matches the expected result. We test our inline function implementation this way. By testing the implementation for multiple languages with significant static semantics differences (think about statement- versus expression languages, object-oriented languages, functional languages, etc.) we show that it can be used in practice and that it can handle edge cases.

We discuss the relevant components of Spoofax in more detail in Chapter 4 (implementation) and Chapter 5 (testing). The interested reader is referred to the Spoofax website[1] for more links to scientific papers on relevant topics.

Contributions. This thesis provides a language parametric implementation of the inline function refactoring. The main contributions are as follows:

- *We investigate challenges that come with inlining a function call that can be generalized* (*Chapter 2*). The challenges are related to preserving the behavior of the program. This mainly concerns changes to the name bindings- and control flow of the program.

[1] `https://www.spoofax.dev/background/`

- *We design a language parametric inline function algorithm* (*Chapter 3*). The algorithm requires a parser that generates ASTs, a static semantics analyzer and language-specific functions as the language parameter. Then, it receives a program and a selected function call, and returns a program in which the function call is replaced by the function body. In addition, the algorithm:
  - Checks that references are correctly retained.
  - Generically solves variable capture and duplicate declarations by renaming where possible. To address variable capture it employs the existing name-fix algorithm [15].
  - Removes the last return keyword before inserting the body, and replaces the function call with the return expression.
  - Exposes to the language engineer potential language-specific issues that cannot be generalized in the form of language parameters. For example, control flow changes are exposed to a language parameter function: it receives the function body and is expected to refactor language-specific constructs that would change the control flow when inlining the function call.
- *We implement our algorithm of function inlining in Spoofax* (*Chapter 4*). The implementation is written in the language Stratego. It can be used by any language defined in Spoofax with a grammar definition in SDF3, a static semantics specification in Statix and a set of language-specific functions defined in Stratego.
- *We test the implementation in Spoofax for three languages using automatic unit tests* (*Chapter 5*). Different edge cases for function inlining are realized in the form of programs in C++, WebDSL and Tiger. Expected output programs where the function call is inlined are compared to the actual output programs of the implementation.
- *We discuss the correctness and usability of the algorithm* (*Chapter 6*). The different steps of the algorithm have dependencies and some are based on previous work with a correctness proof. We provide an informal overview to judge the correctness of the inline function algorithm. We also investigate the usability and runtime performance of the implementation.

Chapter 2

The Inline Function Refactoring

There is more to function inlining than just inserting the function body. As mentioned in the introduction challenges can arise such as changing name bindings because the scope changes. This chapter introduces the challenges that are related to the process of inlining a function itself and apply to many languages, regardless of their syntax or static semantics. Since the challenges apply to many languages, a language parametric algorithm might be able to address them generically. The chapter only focuses on introducing the challenges, the chosen solutions for the algorithm are described in Chapter 3.

2.1 Parameters

Most languages that support function definitions also support function parameters. The arguments to a function call represent the initial values of the parameters. When inserting the function body to inline a function call, the references to the parameters are effectively references to the argument values, which suggests that they can simply be substituted by the arguments as shown in Figure 2.1.

```
1  int foo(int x) {
2    int y = x * 2;
3    int z = x / 3;
4    return y + z;
5  }
6
7  int main() {
8    foo(6);
```

```
1  int main() {
2    int y = 6 * 2;
3    int z = 6 / 3;
4    y + z;
5  }
```

Figure 2.1: Simple example of removing the need for parameter declarations by inlining all occurrences of the parameter.

However, parameters do not have to be constant. In many languages (except for most functional programming languages) you can usually assign new values to the parameters inside the function body. In that case not all occurrences of the parameter can be substituted by the arguments, because it would result in a write to an expression. If the parameter is still forcefully substituted by the argument nonetheless, the invalid syntax displayed in Figure 2.2 occurs.

A simple solution would be to create a variable declaration for every parameter and initialize it with the argument value. This variable declaration can then be placed directly above the inlined code block such that it is reachable. Still, when there is no write to the parameter, this introduces unnecessary overhead of defining a new variable.

Another approach is repeatedly doing what we suggested before: inlining all occurrences of the parameter variable to avoid the overhead of the variable definition. When there are

```
1  int foo(int x) {
2    x = x + 1;
3    return x;
4  }
5
6  int main() {
7    foo(2);
8  }
```

```
1  int main() {
2    2 = 2 + 1;
3    2;
4  }
```

Figure 2.2: An example of issues that could arise with inlining a function because parameters are mutable in C++. Parameter `x` is substituted by the argument value `2`, creating invalid code.

writes to the parameter in the function body, you would have to inline all occurrences after the last write to the parameter, remove the write, and then repeat that process for the remaining writes.

However, even when repeatedly inlining the parameter it is still not always possible to completely get rid of it. Consider the for-loop construct from C++ for example. If a write to the parameter takes place inside the body of the for-loop, we would first have to fully unroll the loop before the inlining process can be applied. To visualize this, think about how it would work for parameter `x` in Figure 2.3. None of the occurrences of `x` can be inlined directly (which in this case would insert the argument value 8) unless the loop is unrolled, because their value changes with every iteration of the loop.

```
1   int half(int x) {
2     for (int i = 0; i < x; i++) {
3       x = x - 1;
4     }
5     return x;
6   }
7
8   int main() {
9     half(8);
10  }
```

```
1   int main() {
2     for (int i = 0; i < x; i++) {
3       x = x - 1;
4     }
5     x;
6   }
```

Figure 2.3: An example of a parameter that cannot be removed by repeated variable inlining. A variable definition is needed for `x` or the program is invalid.

As stated in the introduction of this chapter, the approach we choose to handle parameters is explained in Section 3.4.

2.2 Control Flow

When a computer executes instructions it usually does so from top to bottom, starting from a specific main instruction. There can also be jumps to skip some lines of code or to move to a previous instruction. Together, this order in which the instructions are executed is called the *control flow* of the program.

In imperative languages the control flow of a program is made explicit with the use of specific language constructs. For example, an if-statement can be used to jump over a block of code when the condition evaluates to false, or a function call can jump to a completely different code block in the program. These constructs may depend on the scope and the context in which they are used. Since the scope of the function body changes when a function call is inlined, this can cause changes to the control flow. The return keyword is a great example of this.

2.2.1 The Return keyword

Functions usually return a value. In a language like C++ this value is made explicit with a return statement. The return statement contains the `return` keyword followed by an expression. In the scope of its encapsulating function definition this statement makes sense. It jumps back to the caller returning the outcome of the expression. The keyword is specifically intended to be used in the context of this function definition, so when the function body is moved away from the definition the keyword does not make sense anymore. The function call and definition have a caller-callee relationship that disappears by inlining. Keeping the return statement results in a jump to a higher caller, the one containing the function call itself. Obviously this changes the behavior of the program and has to be fixed.

The purpose of the return keyword, jumping back to the scope of the caller, has disappeared after inlining. It is already in that scope and there is no need to jump back to it. So, we could simply remove the return statement. However, the returned expression represents the value of the function call, so it is still needed. Luckily a function call is an expression too, so the return expression can simply take its place (without the return keyword).

There are two complications with this approach. First, languages like C++ allow void functions that do not return a value. In that case there is no return expression to replace the function call expression. Luckily, a call to a void function means the function call does not represent any value. Because of this, it should not hurt the enclosing expression to simply remove the function call, as long as the referenced statements are still executed. Therefore, the expression containing the function call should still be valid with the function call removed, as long as the statements of the function body have been inserted above the expression.

Another issue with return statements is that functions may contain more than one. If a return statement is located before the end of the function body (we call this an *early return*), the statements below the early return are skipped; the function jumps back to the caller instead. Removing the return keyword removes this jump and thereby changes the control flow, but as discussed above, the return keyword cannot stay either. Many languages do not have an alternative single language construct that can replicate this jump without extra effort, in which case if-then-else branches are needed to replicate the same control flow. The conditionals of the if-then-else branches must represent the conditions in which each early return statement is reached, which can only be derived with a deeper knowledge of the semantics of the language in question, ruling out the possibility of a generic solution.

Our chosen solution for handling the last return and early returns is described in Section 3.5.

2.2.2 Other Constructs

Unfortunately there can be more language constructs that affect the control flow and can thereby complicate function inlining. In fact, theoretically there is an infinite number of possible control flow constructs that have to be taken into account when working generically.

Take the condition of a while-loop for example. It is evaluated before the body of the while-loop is entered and before every next iteration. Suppose that the function call we want to inline is part of the while-loop's condition, see Figure 2.4. For function bodies consisting of multiple statements this poses an issue. The condition only fits the return expression, so the other statements need to be inserted somewhere else. However, as mentioned before, the condition needs to be evaluated at the start of every iteration. Therefore, inserting the function body at the top of the while-loop body does not suffice. The function statements need to be executed before the while condition, which is also executed at the end of every iteration. So the function body needs to be inserted at least twice: once right before the while-loop starts and once at the end of the while-loop's body. Still, we are not there yet, since this does not take additional special constructs into account like the `continue` keyword.

A `continue` statement located before the inserted statements would skip them. Thus, we would need to insert the full function body right before every continue statement as well.

```
1   bool foo(int x) {
2     x = x * 2;
3     return x < 10;
4   }
5
6   int main() {
7     int a = 0;
8     int b = 0;
9     while (foo(a)) {
10      a = a + 1;
11      if (a == 5) {
12        continue;
13      }
14      b = b + a;
15    }
16  }
```

Figure 2.4: Inlining the highlighted function call to `foo` poses several control flow related issues.

On the other hand, some control flow constructs do not complicate function inlining when present in the function body. Consider exception handling for example. If a statement in the function body throws an exception that is caught in the function body as well, inlining the function body does not change the control flow. If the thrown exception is not caught in the function body, then it should be caught somewhere in the scope of the function call, meaning that the statement throwing the exception can still safely be inlined at the scope of the function call.

As long as a language engineer has the complete freedom to come up with new custom control flow constructs, there is no way to cover all of them generically. However, we can still cover the most common control flow construct in the context of function inlining: return keywords. In Section 3.5 we explain how we handle return keywords generically and how we address the other language constructs.

2.2.3 Declarative Languages

Unlike imperative languages, declarative languages are designed to state what is being computed without explicitly specifying the control flow of the computation [16]. Examples of declarative languages are logic- and functional programming languages. You might be tempted to think that the lack of explicit control flow means that you can simply disregard it when inserting the function body, and in many ways that is true. However, declarative languages are always eventually compiled/interpreted to imperative languages, because to execute their programs they have to run on hardware that runs in an imperative manner. This means that there will always be a certain control flow, one that depends on the compiler instead of the language itself. For example, compiled Stratego code executes rules from top to bottom, even though their order of execution is not defined by Stratego because of its declarative nature. Depending on the actions performed by the program, this could affect the observable behavior.

As you can imagine, it is not feasible to generically predict control flow that may be introduced by compilers of declarative languages. Since the control flow of a declarative program is not expressed in the program itself (by design), by definition it should not be affected when moving the function body to the function call (assuming that all other aspects are handled correctly, like valid syntax, name bindings, etc.). Hence, we could actually ignore the control flow of declarative languages. On the other hand, it can still be beneficial to allow a

language parameter to handle control flow exceptions that may apply to a specific language. Section 3.5 mentions how we include this in our algorithm.

2.3 Name Resolution

Programming languages can bind values or code to identifiers, creating variables and functions. The identifiers can then be re-used to reference their bound value or code, creating a reference relation (id_i^R, d_j^D) where id_i^R is the referencing identifier and d_j^D is the referenced value or code. The collection of all such reference relations in a program is called the *name resolution* of the program. Note that i and j are used to differentiate between multiple occurrences with the same name.

Now, which identifier references what not only depends on the identifier that is used, but also on the scope in which the identifier is located. As you can imagine this can pose several issues when inlining a function call.

2.3.1 Which Function Definition?

Although it may seem obvious, finding the function definition belonging to a function call may be more complicated than expected. Firstly, many languages allow function overloading, the definition of multiple functions with the same name. In C++ for example, as long as the parameters are different, nothing is stopping a programmer from defining multiple functions with the same name.

Additionally, functions can be referenced from different scopes, as long as the scope containing the function definition is reachable. Only one of all reachable scopes is the intended scope, even though multiple scopes can contain a definition with the same name. Sometimes the intended scope is determined by prefixed syntax such as when invoking a method on an object. The refactoring should use this prefix to identify the correct scope. Therefore, finding the definition requires an understanding of language-specific syntax like imports and objects that can be used to access other scopes.

Finally, there is the concept of dynamic binding, which includes *mutable functions* and *dynamic dispatch*. In some languages it is possible to call a function without explicitly specifying which function definition is being referenced, in which case the referenced declaration is determined at runtime. Since refactoring is performed at compile time and not at runtime, this makes it impossible to determine which declaration is referenced. Dynamic dispatch for example can occur in multiple forms. In C++ virtual methods or function pointers are examples, in Java method overriding and in Python the built-in `getattr()` function allows for dynamic dispatch. In Section 3.2 we discuss how we find the function definition language parametrically and how dynamic binding is handled.

2.3.2 Reference Relations

If an identifier is moved to a different scope the referenced declaration could change or could even become unreachable. A function call and its corresponding function definition are in different scopes. Therefore inlining the function body may very well change the name resolutions of some of its identifiers.

A simple example of this being an issue is displayed by the inline action in Figure 2.5 in the language C++. Note that the number suffix after identifiers is meant to help differentiate between occurrences of the same identifier, so the name $\texttt{y}_2$ would not be different from $\texttt{y}_1$ in the actual program.

In the example a declaration of variable `y` is inserted ($\texttt{y}_2$), but this clashes with the already existing declaration in the `main` scope ($\texttt{y}_1$). In C++ this duplicate declaration yields an error, which is not behavior-preserving and therefore should be fixed if possible. The issue would

```
1   int foo() {
2     int x = 1;
3     int y₂ = x;
4     x = 2;
5     return y + 1;
6   }
7
8   int main() {
9     int y₁ = foo();
10  }
```

```
1   int main() {
2     int x = 1;
3     int y₂ = x;
4     x = 2;
5     int y₁ = y + 1;
6   }
```

Figure 2.5: An example of what can go wrong when moving a variable declaration from one scope to another. Inlining the function call `foo()` on the left results in a duplicate declaration of `y` on the right.

be resolved by getting rid of the inlined declaration of y_2. A way to achieve that is by renaming the inserted declaration and all its intended references to a new unique name.

Next to conflicts in the same scope, *shadowing* can also mess with the name resolution of the program. Shadowing means the re-declaration of an identifier in an inner scope relative to the already existing declaration. In most languages shadowing is allowed, so it should not produce errors like duplicate declarations do. However, it can still change the behavior of the program. Suppose a declaration x_2^D in the function body has the same name as a declaration x_1^D in an outer scope relative to the function call. Inlining the function will result in x_2^D shadowing the existing declaration x_1^D. This could change a reference relation $(x_i^R,\ x_1^D)$ to $(x_i^R,\ x_2^D)$ of any referencing identifier x_i^R (see Figure 2.6). If this happens, the reference relation $(x_i^R,\ x_1^D)$ is captured by declaration x_2^D changing the behavior of the program (Note that again the number suffix is only added to differentiate between occurrences of the same identifier). A possible fix would be similar to the fix for duplicate declarations: renaming x_2^D and all of its intended referencing identifiers in the function body.

```
1   int foo() {
2     int x₂ = 1;
3     return x + 1;
4   }
5
6   int main() {
7     int x₁ = 5;
8     if (x < 6) {
9       foo();
10      x = 6;
11    }
12  }
```

```
1   int main() {
2     int x₁ = 5;
3     if (x < 6) {
4       int x₂ = 1;
5       x + 1;
6       x = 6;
7     }
8   }
```

Figure 2.6: Inlining `foo()` yields variable capture by declaration x_2^D, changing the behavior of the program.

However, there are plenty of cases where shadowing will not change the behavior of the program. As long as an inserted declaration x_j^D shadowing x_k^D does not break any reference relations outside the inserted code block $(x_i^R,\ x_k^D)$ and shadowing is allowed by the static semantics of the target language, inserting the declaration in question does not capture any references and therefore does not change the behavior of the program. An example of this can be seen in Figure 2.7. Although the inserted declaration x_2^D shadows the declaration x_1^D in `main()`, x_1^D is only referenced in outer scopes relative to the function call. Since shadowing x_1^D in the scope of the function call does not capture a reference relation, there is no need to apply any changes.

There are still other situations left in which the name resolution of the program changes.

```
1   int foo() {
2     int x2 = 1;
3     return x + 1;
4   }
5
6   int main() {
7     int x1 = 5;
8     if (x < 6) {
9       foo();
10    }
11    x = 6;
12  }
```

```
1   int main() {
2     int x1 = 5;
3     if (x < 6) {
4       int x2 = 1;
5       x + 1;
6     }
7     x = 6;
8   }
```

Figure 2.7: An example of shadowing occurring after inlining. In this case it does not change the behavior of the program, so it is allowed.

Assume a reference relation $(\texttt{x}_i^R, \texttt{x}_j^D)$, where $\texttt{x}_i^R$ is inside the function body and $\texttt{x}_j^D$ is not. Now assume that from the scope of the function call it is not possible to reference $\texttt{x}_j^D$ with $\texttt{x}_i^R$ without adding some prefixed syntax to $\texttt{x}_i^R$. Take the class method `addX` in Figure 2.8 for example. Here attribute $\texttt{x}_1$ is $\texttt{x}_j^D$ and `x` in method `addX` is $\texttt{x}_i^R$. Inlining the call to `addX` outside the class without changing the syntax of `x` will break the reference relation, so a fix is needed. Luckily, adding the prefix `myObj.` restores the intended reference relation.

```
1   class MyClass {
2     public:
3       int x1 = 5;
4
5       int addX(int y) {
6         return x + y;
7       }
8   };
9
10  int main()
11  {
12    MyClass myObj;
13    myObj.addX(5);
14  }
```

```
1   class MyClass {
2     public:
3       int x1 = 5;
4
5       int addX(int y) {
6         return x + y;
7       }
8   };
9
10  int main()
11  {
12    MyClass myObj;
13    int y = 5;
14    x + y;
15  }
```

Figure 2.8: An example of a reference that breaks after inlining, but that can easily be fixed by changing the identifier to `myObj.x`.

Unfortunately restoring the reference relation by changing the identifier is not always possible. Some declarations are simply *unreachable* from the scope of the function call. If the attribute `x` in Figure 2.8 were private instead of public, it would be unreachable from the scope of the function call. Adding a getter method `get_x()` to `MyClass` that returns `x` could be a solution, but usually attributes are private for a reason, so we must be careful about adding a getter.

Chapter 3

Language Parametric Function Inlining

The aim of this thesis is to define the function inlining refactoring language parametrically. The previous chapter introduced challenges with inline function. This chapter describes how we tackled those challenges and shows the algorithm we designed. The structure of the chapter is as follows: first, it lists the prerequisites of the algorithm, the language parameters (Sections 3.1 and 3.2). Then, the approach to each of the challenges from the previous chapter is described (Sections 3.3 to 3.7). Finally, we give an overview of the full algorithm in the form of pseudocode (Section 3.8).

3.1 A Structured Program Representation

3.1.1 Parser

The first language parameter is a syntax parser for the target language. This parser must convert string representations of programs to Abstract Syntax Trees (ASTs). An AST is similar to a parse tree, but abstracts away syntactical details such as comments, whitespacing, or other formatting characters like ";" in C++. ASTs are often used as an Intermediate Representation of a program, for example by many compilers [10] but also by (automated) transformations to programs.

For a generic algorithm, it must be possible to explore the AST generically. To be able to do this, the nodes of an AST need to have a language independent structure. This is achieved by using Annotated Terms (ATerms) for AST nodes, as is done by SDF3 [9]. An ATerm can be an integer, a string, a list of ATerms, a tuple of ATerms, an ATerm with an annotation (the annotation is also an ATerm), or a constructor application of ATerms. A constructor application is an identifier (the constructor name) followed by a sequence of ATerms.

The root node of an AST (which holds the full tree) represents a complete program and its child nodes are components of the program. For example, a possible AST term representing the function call `foo(2, "h")` can be seen in Figure 3.1. Without an AST encoding, finding and isolating components of the function call, for example the argument expressions, requires a traversal through its string representation. You would have to check substrings or even individual characters based on the syntax of the language in question, which essentially translates to writing a parser anyway. Therefore, it makes sense to ask for a full AST parser as a language parameter.

A grammar or syntax definition of a language usually already specifies many syntactical details of a language. This way, the definition implicitly defines part of a parser that translates program strings to AST nodes. For example, consider the Syntax Definition Formalism (SDF

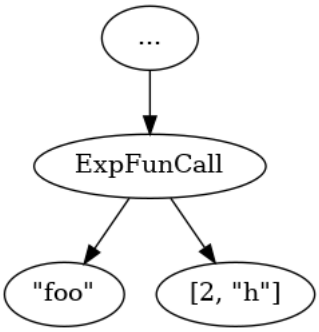


Figure 3.1: A possible AST node representation of the function call `foo(2, "h")`.

[17, 9]) rule of a function definition below. It belongs to a grammar, but it can identify if-statements and parse them to AST nodes with `<Exp>` and `<Statement>` as child nodes.

```
Statement.If = <if(<Exp>)<Statement>>
```

In fact, with the addition of disambiguation- and layout rules it is possible to generically derive an AST parser from the syntax definition [9]. Since a syntax definition is required for a language to exist, this makes the requirement of a syntax parser a lot less demanding.

3.1.2 Unparser

Next to a parser we need something that can do the opposite. After all, once the AST has been transformed, we need to return the program to the user in the form of a source code string in the target language. This comes down to a applying an 'unparser' to the AST, which converts the AST to a valid program string.

However, the parser has abstracted away syntactical sugar with the creation of the AST. This means that by unparsing the AST, certain layout and formatting from the original program would disappear, such as comments and whitespacing. Fortunately, De Jonge and Visser recognized this issue in the past [18] and have designed an algorithm that unparses a transformed AST whilst preserving the layout and formatting of the original program, including comments. The algorithm is language-parametric and does not require the parser to store additional information in the AST. Instead, it compares the original AST and the transformed AST to determine which nodes of the AST have changed. Nodes from the original AST are linked to substrings of the source code by "origin tracking", a method developed by Van Deursen et al. [19]

This way, only the program substrings related to transformed nodes have to be changed. Changed nodes are unparsed by a regular unparser, but where possible the algorithm still attempts to preserve layout, for example by examining how the original program attaches certain sequences of nodes layout-wise (the location of the semicolon between statements in C++ for example). As a result, layout outside the transformed substrings is not lost and the layout of the changed substrings corresponds to the original layout as much as generically possible. To apply the layout preservation algorithm we do need a regular unparser for the transformed nodes, but similarly to the parser it is possible to derive this from the grammar with the presence of disambiguation- and layout rules [9].

3.1.3 Language-Specific Strategies

Many ATerms are constructor applications and have a label (the constructor identifier) indicating what kind of node it is. Based on the grammar, constructors with this label must have a specific set of child nodes in a specific order. A language engineer is free to design their own grammar and can therefore choose anything they want for the constructor names,

the amount- and order of child nodes, etc. Now, language constructs like a function call expression usually have a similar structure regardless of the language: a function call contains a string holding the name of the function that it refers to, followed by argument values that are expressions themselves.

However, it is not possible to retrieve this information from the AST node without knowing its structure. Look at Figure 3.2 for example. It shows a C++ (left) and Tiger (right) ATerm of a function definition that receives an integer `x` and returns `x + 1`. As you can see the ATerms have a vastly different structure. Consequently, from a language independent perspective we do not know which subterm represents what part of the function definition.

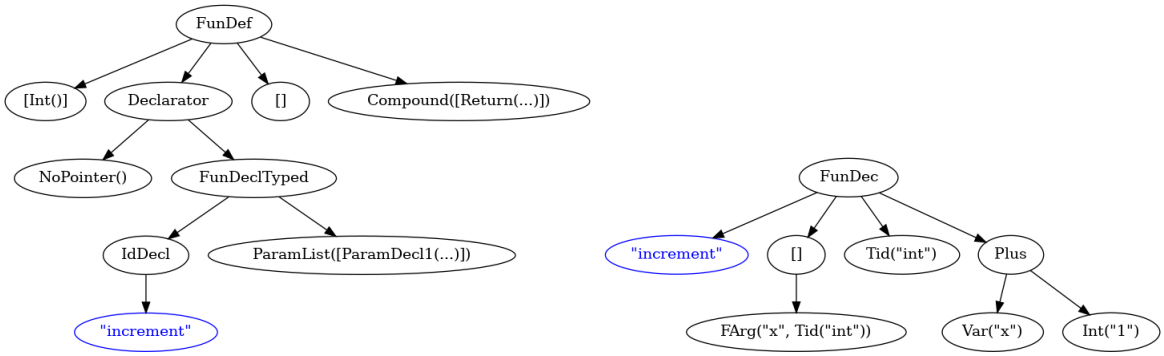


Figure 3.2: The ATerm on the left is from an existing C++ grammar, the ATerm on the right is from a Tiger grammar. Both ATerms represent a function definition that receives an integer and increments it by 1, but have vastly different subterms.

Therefore, we require the language engineer of each language to define helper functions that can extract the required information. Keep in mind that these are not complex functions if you know the grammar of the language in question. For example, to extract the parameter nodes all that is required is a function that receives a function definition ATerm and returns the parameter subterms, which can be implemented by a simple pattern match. From this point on, functions or strategies that have to be implemented by the language engineer will be referred to as *language-specific strategies*. For an overview of each language-specific function we require including a description of their purpose, see Section 4.4.

3.2 Static Semantics Analyzer

When a user wants to apply the refactoring in practice, they select a function call expression in the source code. The selected expression is represented by a function call node in the AST. The first step of inlining the function call is finding the corresponding function definition node in the AST. This might sound simple: Explore the AST until a string node is found matching the function name, where the enclosing node must be the function definition. But, what if another call to the function is encountered before the definition? Or, what if there are multiple definitions with the same name? (Section 2.3.1) Instead of blindly matching on the name string we need to take certain properties of the nodes into account relating to the static semantics of the language. For a function call, the referenced node needs to have the function definition label from the grammar. Furthermore, depending on the language, the arguments in the call need to match the parameters in the definition, both in quantity and in type. Then we still have to check the priority of scopes, reachability, imported files, etc.

All these language properties are defined by the semantics of the language. Together, they decide the *reference relation* between the identifier and its definition. Reference relations are relevant to all languages — using identifiers to reference declarations is a universal concept — so if the reference relations are known to our algorithm, we should be able to query the

reference relation of the function call language parametrically. Therefore, we need a static semantics specification for the language in question. This specification should be able to statically analyze programs and derive the reference relations. Then, the derived relations should be accessible with a simple API call. Because a reference relation not only depends on the identifier string but also on its scope, we expect the analyzer to add AST annotations such that it can distinguish between multiple nodes with the same name. Then to query the reference relation of the call node from Figure 3.1 the API call needs to be invoked on child node `"foo"` and its unique annotation.

Because the static semantics specification analyzes language-specific properties of a program, it should also notice violations of the static semantics that may arise by transforming the AST. Examples include when an identifier is unable to reference any declarations from a scope, or duplicate declarations in C++. We expect the static semantics analyzer to detect and report such errors, next to deriving reference relations.

If the single use case of finding the function definition is not sufficient motivation for a dependence on a static semantics specification, know that more reference relations are explored throughout function inlining. For example, consider the checks that have to be performed to ensure that the reference relations of the program are retained (Section 2.3.2).

Note that it is not possible to derive reference relations from the static semantics if the language in question allows dynamic binding, i.e., it allows the use of references that cannot be determined at compile time. This includes the dynamic dispatch examples described in Section 2.3.1. Another example is the `getattr(object, name)` function in Python, which allows a programmer to reference the method of an object by providing its name as a string variable. The contents of the string variable are unknown at compile-time, so the referenced method is unknown too. Since it is not possible to derive all reference relations using the static semantics specification, our function inlining implementation cannot be used when a program contains instances of dynamic binding.

A static semantics analyzer may return a list of possible declarations to deal with dynamic binding. In that case our algorithm can still be applied, as long as the reference lists we receive contain exactly one declaration. Else, we have to abort the inlining action. This not only applies to finding the function definition that belongs to the function call, but also to any other reference relation we explore throughout the algorithm.

3.3 Extracting the Required Nodes

Data from the AST is required to correctly inline a function call. The algorithm starts with only the full AST and the function call node. The first thing we need to extract is the child node holding the function identifier in the call node. This identifier will be used to query the reference relation as described in the previous section. However, which child node holds the identifier depends on the design of the parser, as discussed in Section 3.1. A language-specific strategy is required to extract the identifier. Similarly, we need the argument expressions from the function call to instantiate the parameters (more details on this in the next section). Because the function call encapsulates both nodes, a single language-specific strategy that returns a tuple holding the identifier and the arguments is sufficient.

Now that the function call identifier has been extracted we can get the function definition identifier from its reference relation. This should be a child node of the full function definition node. But, we also need the child node holding the body, the node holding the parameters and (depending on the language) also the node holding the function type. Another language-specific strategy is required to go from the identifier to these other child nodes. The strategy should simply know their position relative to the identifier node, which is based on the grammar of the language in question.

The two language-specific strategies described above are easy to implement for a language engineer, because the constructor design is known. For example, consider the function definition structure displayed in Figure 3.3. It is a function definition for `foo`, which in this example has return type `Int` and two parameters. As described above, the algorithm finds the identifier node by querying the static semantics analysis (the colored node `"foo"` in Figure 3.3). Extracting the body from there is simple if you know the structure: Go up one node, then take the fourth child node.

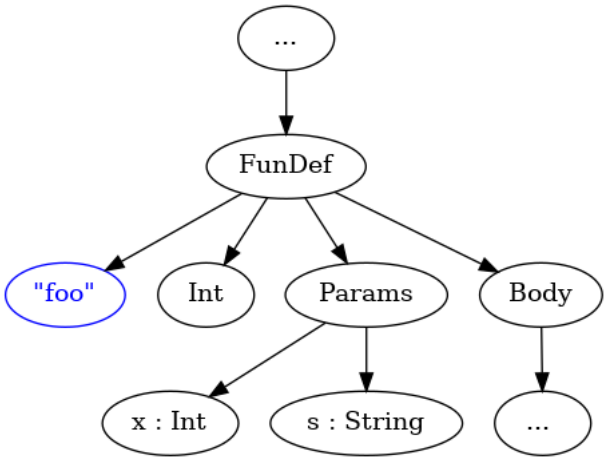


Figure 3.3: An example of a possible structure design of a function definition node in an AST.

3.4 Instantiating Parameters

The issue with parameters has been introduced in Section 2.1. For our algorithm have chosen to create a new variable definition for every parameter. This means that unused parameters or parameters whose value never changes also receive a definition. The main argument for our choice is that we consider inlining the parameter declarations to be a different refactoring. It is not up to us to decide that the extra refactoring improves readability. Furthermore, if we apply extra refactorings here, then what is stopping us from applying other simplifying transformations as well? Consider Figure 3.4 for example, where inlining parameter `x` yields unreachable code that can be reduced by a remove-unreachable-code refactoring.

```
1   int foo(bool x) {
2     if (x) {
3       return 0
4     }
5     return 1;
6   }
7
8   int main() {
9     foo(false);
10  }
```

```
1   int main() {
2     int result;
3     if (false) {
4       result = 0;
5     };
6     result = 1;
7     result;
8   }
```

Figure 3.4: Inlining a call to `foo` with `x = false` yields dead code, which makes the remove dead code refactoring applicable.

Especially once you start applying refactorings like constant folding/propagation, many other refactorings could become applicable too. Because the focus of this thesis lies on the function inlining refactoring on its own, we choose to not reduce the parameter definitions in the algorithm.

Now to create a variable definition for a parameter you need at least two things: the identifier and a value. Many languages also require the type of the defined variable. The identifier and its type are extracted from the function definition, the values are extracted from the function call (as described in Section 3.3). Then a new variable definition node has to be constructed. Since the structure of a variable definition node is language-specific we require another language-specific strategy that fills in the definition node. By applying this to all parameter nodes we end up with a list of variable definitions.

3.5 Dealing with Return Statements

As stated in Section 2.2.1, we cover return statements in our algorithm because they are the most common control flow construct in the context of function inlining. Not handling them would severely limit the number of function definitions to which our algorithm can be applied.

Return statements abruptly return to the caller of a function when executed. Inlining them without removing them is not behavior preserving. Additionally, they contain a return expression that must replace the function call. As explained in Section 2.2.1, in case of a single return statement at the end of the function body the solution is simple: replace the function call with the return expression and remove the return keyword. For void functions with only a single return statement (or no return statements) the solution is similar, but the function call is not replaced by an expression. Instead, the function call has type void, so it can simply be removed.

Finding the last return statement is a language-specific task. The function body has to be traversed whilst knowing if a visited node is a return statement or not. Furthermore, the order of execution of the statements needs to be known, since the statement that will be executed last needs to be found. For example, if for whatever reason a language developer decides that statements are executed from bottom to top, the statement executed last will be the first node in the function body AST node.

Early returns, return statements before the end of the function body, are not so easy to deal with. They need to be removed, but then the jump to the end of the function body has to be replicated using other language constructs. This is not possible to achieve generically: it requires a deeper knowledge of the language in question, mainly about the control flow semantics of the language. Now, we could take the same approach as with reference relations, where we assume the existence of a static semantics specification for reference relations. Similarly, a specification that generates a Control Flow Graph (CFG) would be useful here. It might be possible to define a language parametric *control flow generator* that can derive the necessary if-then-else conditions by exploring the edges of the original CFG, but this concept has not been worked out yet and is out of scope for this thesis. We do mention it as a possible improvement to our algorithm in Section 8.1.

Instead, we ask the language engineer to remove early return statements with a language-specific strategy. This strategy receives the function body and should return a tuple containing a return-less function body and the return expression. In case the language engineer decides not to go through the effort of removing early returns and they are still present in the returned body, the algorithm will abort. In that case the algorithm only works for functions without early returns.

The added benefit of doing it this way is that the language engineer now has a language-specific function that can alter the function body. If there are any other (control flow) issues that only apply to the language in question, they can be dealt with in the same function. Now, this does make it the responsibility of the language engineer to ensure that the control flow is unchanged, but without other language parameters like a CFG generator it is impossible to generically check that anyway, so it is better than nothing.

3.6 Transforming the AST

With the function parameters and body prepared for inlining, it is time to insert them into the AST. So far, the fact that the AST structure is different for every language has mainly been an issue for extracting information. However, now that we need to insert code, we need to adhere to the AST structure of the language in question. For expression languages like the toy language Tiger [20] this is simple. In Tiger, everything is considered an expression, from single values to assignments to a full program. Naturally, this means that it has a language construct in which the parameter assignments and the function body can be combined to form a single expression. This expression can simply replace the function call expression to successfully inline the function call.

However, for languages that distinguish between statements and expressions (C++ for example) it is not that simple. In those languages an expression like a function call is part of a statement, which in turn can be part of a sequence of statements. If we want to replace the function call expression it has to be by another expression, not a statement. The function body on the other hand consists of a statement or a sequence of statements. To visualize this complication, an AST encoding of a function call statement can be seen in Figure 3.5 and similarly a function body in Figure 3.6.

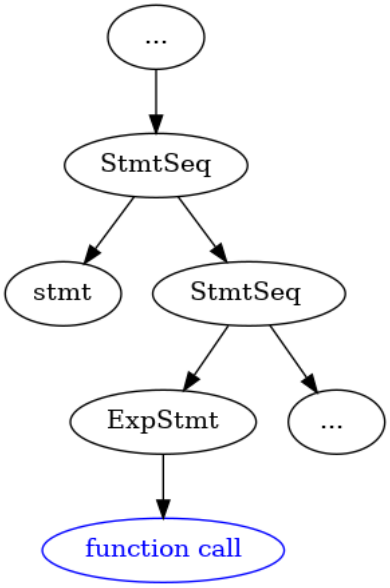


Figure 3.5: An example of the sequence of statements wrapping a function call in an AST.

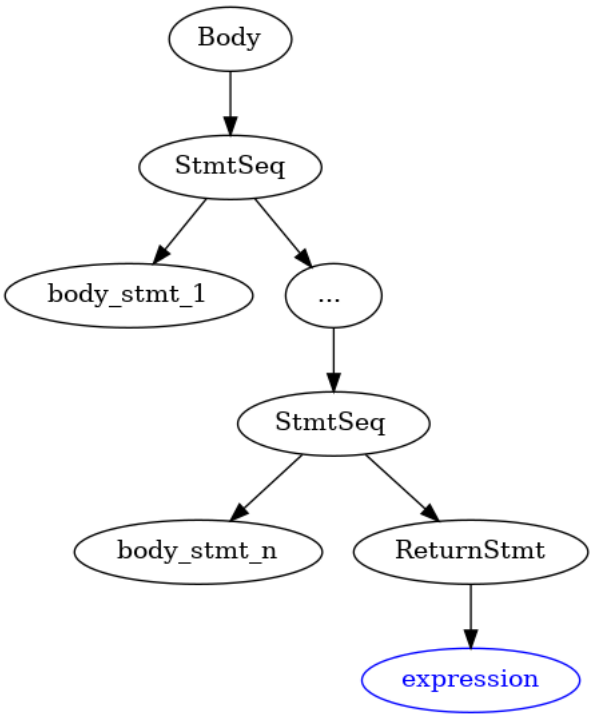


Figure 3.6: An example of the sequence of statements in a function body node in an AST.

One of the body statements contains the expression value that is returned to the function call. As explained in Section 3.5 we clear the function body of return keywords and other constructs that could change the behavior of the program with a language-specific function. This function also extracts the return expression, so we can safely use that to replace the function call expression (in Figure 3.6 this expression is colored blue). Next, we need to insert the other body statements, since the return expression only evaluates to the intended value if they have been executed first. They need to be inserted between the statement containing the function call and the previous statement. The inserted body statements also need to respect the AST structure of the language. For the example AST components in Figure 3.5 and

Figure 3.6, this yields the new AST displayed in Figure 3.7 where the inserted and changed nodes are colored blue.

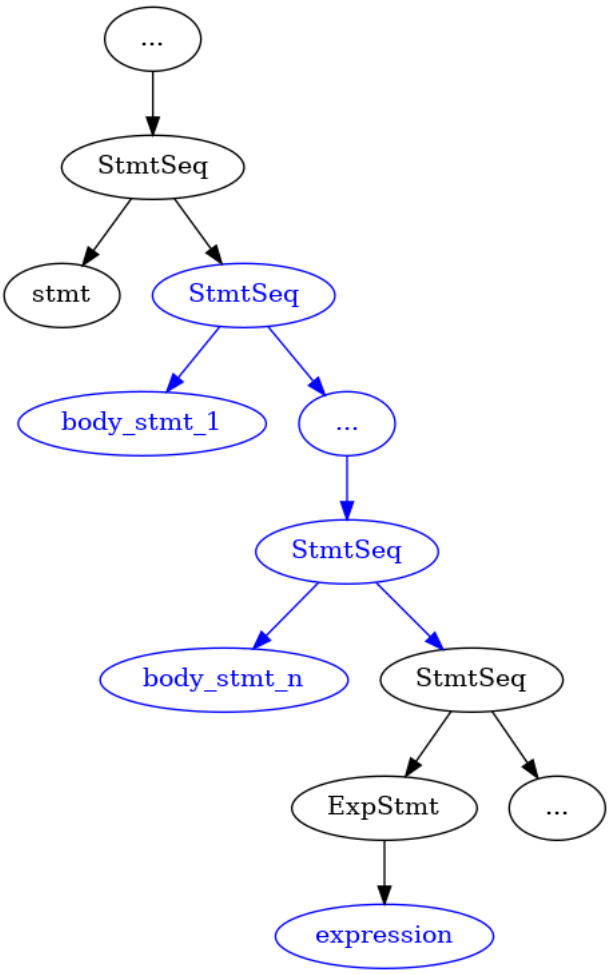


Figure 3.7: The function call of Figure 3.5 has been inlined with the body from Figure 3.6. The blue nodes and edges indicate what has changed.

The parameter definitions created in Section 3.4 must be inserted above the body statements. Variable definitions are statements too, so they can be added similarly to the body statements.

Unfortunately constructor names like `StmtSeq` are not generic, the language designer is free to choose their own. Similarly, a sequence of statements is not necessarily defined in a linked list manner. For example, it can also be defined as a single node containing a regular list of statement nodes. Consequently, it is impossible to completely generalize the process described above. To a generic implementation it is unknown how many nodes we have to traverse up from the function call to reach the enclosing statement as it can be nested in many other expressions. It is also unknown how we insert a sequence of statements into another sequence since we do not know its structure. Because we require language-specific knowledge, we have to ask for language-specific strategies.

The simplest approach would be to ask for a language-specific strategy that receives the AST, function call, parameter declarations and body, and transforms the AST accordingly. For a Tiger grammar this can be a trivial strategy: creating a node `Let(parameter declarations, body)` and substituting it for the function call in the AST. For languages with statements this process is more involved, but it can be summarized as follows:

1. Find the statement containing the function call expression.

2. Get the sequence of statements containing the function call statement.
3. Split the sequence at the function call statement.
4. Convert the parameter declarations to a sequence of assignment statements according to the statement sequence constructor of the language.
5. Combine the parameter declaration statements, the body statements and the function call statement according to the statement sequence constructor of the language in question.
6. Re-attach the split sequences from 3.
7. Replace the function call expression with the return expression. In case the function is of type void (or a similar type) remove the expression without replacing it with anything.

It might be possible that the statement containing the function call is not part of a sequence of statements. In that case step 3 and 6 can be skipped.

Now, step 1 can be performed generically given a language-specific function that can recognize statement nodes. Steps 4 and 5 can be performed with a language-specific function that receives the statements and combines them into a single sequence. Then steps 2, 3 and 6 can be combined into a single language-specific function that receives the code block and the enclosing statement, and inserts the block in front of the statement. We can perform step 7 generically since we already have the function call and return expression.

It is the responsibility of the language engineer to deal with any language-specific edge cases when inserting the code block. An example of such an edge case: in C++ the function call can be part of a for-loop condition as displayed in Figure 3.8. The update step of a for loop in C++, in which the function call to `increment` is located, cannot hold multiple statements. The body therefore has to be inserted somewhere else. Because the body needs to be executed at the end of every iteration, it should be inserted at the end of the loop-body. However, then the `continue` statement would skip it, meaning we also need to insert the body directly before the `continue` statement. The language engineer can of course decide to disallow function inlining in this situation, causing the algorithm to abort and return the original program. Otherwise, it is their responsibility that the control flow is preserved by the insertion.

```
1   int increment(int x) {
2     int y = 2;
3     x = x * y;
4     return x + 1;
5   }
6
7   int main() {
8     for (int i = 0; i < 10; i = increment(i)) {
9       if (i % 2 = 0) {
10        continue;
11      }
12      doSomething();
13    }
14  }
```

Figure 3.8: An example of a function call that is difficult to inline due to its surrounding statement. Inserting the body of `increment` in the for-loop while preserving the behavior of the program is problematic.

3.7 Reference Validation

After the AST is transformed, it should be analyzed by the static semantics specification. This extracts the new scope graph and detects potential static semantic- or syntax errors. Syntax errors are not expected, if they occur it means that one of the language-specific strategies produced invalid AST nodes for the language in question. Semantic errors on the other hand can still occur even if the language-specific functions are implemented correctly. Since semantic errors returned by the analyzer are language-specific we cannot fix them directly in our algorithm. Instead, we expose them to the language engineer in a new language-specific strategy. It is up to the language engineer to decide which errors to fix in this strategy, after which we re-analyze the returned AST only if the AST has been changed.

Once all static semantics errors have been resolved, it is time to perform to reference validation. The sections below explain our approach to reference validation.

3.7.1 Introducing Name-Fix

In Section 2.3 we described that the reference relations of a program can change when the function body is inserted in the lexical scope of the function call. Changes to reference relations can change the behavior of the program without triggering semantic errors. To ensure that the transformation is behavior preserving, we need to detect and fix these changes.

In the language parametric refactoring "Renaming" by Misteli [21] a reference validation strategy is also employed. His strategy is based on the *name-fix* algorithm [15]. Name-fix is intended to detect (*variable*) *capture* resulting from applying a transformation to a program. It detects most issues described in Section 2.3 where reference relations change. In this section we explain how we employ the name-fix algorithm to detect capture after inlining a function.

3.7.2 Detecting Capture

The name-fix algorithm compares the *name graph* of a program before and after a transformation. The paper on name-fix defines a name graph as a tuple of a set of IDs V and a relation ρ. Each referencing- and declaring identifier in program p is assigned a unique ID v, and the set of all such IDs is V. The reference relations of a program are denoted by the partial relation $\rho \in V \rightarrow V$, where $\rho(v_r) = v_d$ if and only if v_r references v_d.

In our case, the name graph of a program can be derived using the language parameters. First, we loop through the AST, generating a unique ID v for every identifier x. Since we can query the reference relations of identifiers, we can loop through the AST and check if an identifier with ID v_r references a declaration. If a referenced declaration is returned, we add its ID v_d to ρ such that $\rho(v_r) = v_d$.

Now that we can derive name graphs of the program before- and after inlining the function call (let us call them G_s and G_t respectively), all relations that indicate capture have to be detected. The paper on name-fix [15] defines a transformation to be capture-avoiding as follows (note that $dom()$ returns the domain of a relation):

Definition 1. A transformation $f : S \rightarrow T$ is capture-avoiding if for all $s \in S$ with $G_s = (V_s, \rho_s)$ and $t = f(s)$ with $G_t = (V_t, \rho_t)$:

1. Preservation of reference intent: For all $v \in dom(\rho_t)$ with $v \in V_s$,
   (i) if $v \in dom(\rho_s)$, then $\rho_s(v) = \rho_t(v)$,
   (ii) if $v \notin dom(\rho_s)$, then $v = \rho_t(v)$.
2. Preservation of declaration extent: For all $v \in dom(\rho_t)$, if $v \notin V_s$, then $\rho_t(v) \notin V_s$.

An observant reader might see a problem with this definition. Assume that a referencing identifier v_r in the function body references a declaration v_d outside the function definition such that $\rho_s(v_r) = v_d$. After inlining the function call, a new node with ID v'_r has correctly been inserted such that it references the same declaration. This yields $\rho_t(v'_r) = v_d$ with $v'_r \notin V_s$ and $\rho_t(v'_r) \in V_s$. However, according to point 2 of Definition 1 we need $\rho_t(v'_r) \notin V_s$, a contradiction. Name-fix handles this as follows. Whenever a transformation copies an identifier x to create x', the ID corresponding to x is copied as well and applied to x'. This way, the new references and declarations created for function inlining (including parameter declarations) have the same ID as the corresponding references and declarations in the original program. This means that in the counterexample $v_r = v'_r$ and therefore $v'_r \in V_s$, fixing the violation of Definition 1.2.

With Definition 1 at our disposal we can check the name graphs G_s and G_t for capture. Any $v \in dom(\rho_t)$ that violates it is stored together with $\rho_t(v)$. If no such pair is found, we can conclude that there is no capture.

Note that if an inserted reference is captured by a different declaration, ρ_t now maps its ID to two declarations (the inserted reference has the same ID as the original reference from which it was copied. The original reference should still reference the original declaration in ρ_t). In an implementation of the algorithm you should be careful to not only check that ρ_t still maps original IDs to their original declarations, but also that it does *not* map them to other IDs at the same time.

Note also that Definition 1.2 requires that $v \notin V_s$. For our inline function algorithm this will never be the case: Copied identifiers receive the same ID as their original counterpart, and function inlining only copies identifiers, it never generates new ones. Therefore, we do not have to check Definition 1.2.

3.7.3 Fixing Capture

After detecting name capture we should have a list of reference relations $(v, \rho_t(v))$ in program t where v has been captured by $\rho_t(v)$. Name-fix will fix all of these capturing declarations by renaming $\rho_t(v)$ and all $\{v | v \in dom(\rho_s), \rho_s(v) = \rho_t(v)\}$. The new name that is given to a declaration and its intended references should be unique to avoid introducing other naming conflicts.

There is one key difference here between the refactoring function inlining and renaming when it comes to fixing references. Renaming identifiers does not change their scope, whereas inlining a function call moves many identifiers from the function definition to a different scope. Because of this, two issues with function inlining described in Section 2.3 are not present for renaming. Firstly, a moved identifier might need some added syntax to reach the intended declaration. Secondly, a declaration can be unreachable from the scope that its references are moved to. Both of these issues might not be detected by the name-fix algorithm: either $v \notin dom(\rho_t)$ because the reference could not be resolved, which means the "For all" clause does not apply (For all $v \in dom(\rho_t)$ ...), or another declaration has captured v resulting in $\rho_s(v) \neq \rho_t(v)$ violating Definition 1.1(i). When another declaration captures the reference name-fix will rename the capturing declaration, whereas it should be fixed by adding a prefix to the captured reference. Figure 3.9 demonstrates a situation where this would occur. However, by repeatedly applying name-fix all the capturing declarations will be renamed, which means eventually only the first case remains where $v \notin dom(\rho_t)$.

Unfortunately the two issues described above are not fixed by giving the relevant identifiers a new unique name. The first issue was that a referencing identifier requires some added syntax to preserve its reference. This either causes an 'unresolved reference' error by the static semantics analysis, or it is captured and renamed by name-fix until it becomes an 'unresolved reference'. Both ways lead to failure. The reference can only be fixed by generating the correct syntax to reach the intended scope. A semi-generic algorithm that generates

```
1   class MyClass {
2     public:
3       int x1 = 5;
4
5       int addX(int y) {
6         return x + y;
7       }
8   };
9
10  int main()
11  {
12    MyClass myObj;
13    int x2 = 2;
14    myObj.addX(5);
15  }
```

```
1   class MyClass {
2     public:
3       int x1 = 5;
4
5       int addX(int y) {
6         return x + y;
7       }
8   };
9
10  int main()
11  {
12    MyClass myObj;
13    int x2 = 2;
14    int y = 5;
15    x + y;
16  }
```

Figure 3.9: The blue method call `myObj.addX(5)` is being inlined. Name-fix will try to fix capture by renaming x_2 whereas it should prefix the red `x` with `myObj.` to reach the scope of the class.

such a reference is under development at TU Delft, but the strategy is still under development so we include it in future work (Section 8.1), and our algorithm does not fix this issue (yet).

The second issue, where the declaration is unreachable, cannot be fixed at all (because it is unreachable) meaning the function inline action should be rejected. The fact that it is unreachable is discovered by an 'unresolved reference' error from the static semantics specification, similarly to the first issue.

3.7.4 Inlining Recursive Calls

There is one type of function call where name-fix does not detect capture: recursive calls. Even though it might seem impractical to inline a recursive function call, it could be part of a bigger refactoring, for example a refactoring that performs a sequence of inlining and constant folding refactorings in an attempt to get rid of a recursive structure. For completeness, we also have to cover recursive function calls in our algorithm.

Figure 3.10 shows the result of inlining a recursive call in the current state of our algorithm. A declaration has been created for the parameter, but this declaration captures three references from the original program (lines 10 and 11). Normally name-fix would discover that the existing three identifiers now reference a declaration with a different ID, but since copied declarations receive the same ID as the original, name-fix does not detect it.

However, we know that an inserted declaration should only be referenced by other inserted identifiers. After all, its sole purpose is exposing the declaration to the inserted identifiers. If an inserted declaration does capture a non-inserted identifier, we know that something went wrong. We can easily fix it by renaming the inserted declaration and all corresponding inserted references with the knowledge of ρ_S from name-fix. Now, this does require us to be able to differentiate between inserted and non-inserted identifiers. To achieve this, we annotate the inserted nodes with `"inlined"` before inserting them into the AST. Then, if an identifier without the annotation references a declaration v_d with the annotation, we rename the declaration v_d and all references with $\rho(v) = v_d$ that *also* have the `"inlined"` annotation. After fixing all inserted declarations, we have to re-analyze the AST and re-apply name-fix. The renaming step is essentially the same as what name-fix does with the only exception that it checks for the `"inlined"` annotation before renaming a string node.

This process has some overlap with name-fix. If a non-inserted identifier originally referenced a declaration outside the function definition and is now captured by an inserted

```
1  int foo(int x) {
2    if (x < 10) {
3      x = x + 1;
4      x = foo(x);
5      x = x * 2;
6    }
7    return x;
8  }
```

```
1  int foo(int x) {
2    if (x < 10) {
3      x = x + 1;
4      int x = x;
5      if (x < 10) {
6        x = x + 1;
7        x = foo(x);
8        x = x * 2;
9      }
10     x = x;
11     x = x * 2;
12   }
13   return x;
14 }
```

Figure 3.10: Inlining the recursive call to `foo` creates a declaration for the parameter (first red `x`) that has the same ID in name-fix, but that declaration captures three existing occurrences of `x` (the other red ones).

declaration, this new step and name-fix would both detect and fix it. This should not be an issue however, it just means that some capture cases are double-checked.

3.7.5 Duplicate Declarations

Finally, there is the issue of duplicate declarations. Since it is an example of a static semantics error, the static semantics analysis should detect it before name-fix is applied. However, a duplicate declaration is essentially a declaration capturing another declaration. As you can imagine, this is very suitable for fixing by renaming similarly to what name-fix does: we take the ID v_d of the declaration that duplicates another declaration, we explore ρ_s to find all IDs that should reference v_d and then we rename them. But, this also renames identifiers in the function definition because they share their IDs with inserted identifiers. Luckily, duplicate definitions can only be caused by the inserted declarations from the inline action: no other declarations are inserted and renaming does not introduce new duplicate declarations since we rename to unique names. Therefore, when fixing duplicate definitions we only have to rename identifiers annotated with `"inlined"`. Since this is a generic process, we offer fixing a duplicate declaration as a helper function that can be invoked by the language-specific strategy that deals with (static) semantic errors, to make life easier for the language engineer.

3.8 Algorithm

The previous sections in this chapter describe the steps that the function inlining algorithm has to perform. All the sections combined yield the full algorithm. Algorithm 1 below contains the pseudocode for our algorithm. Each step in the algorithm starts with a comment that explains which step it is. Language-specific strategies are prefixed by `ls` to point out which functions cannot be implemented generically.

To fit the pseudocode on a single page the pseudocode for reference validation and name-fix is located in Algorithm 2. The idea is that Algorithm 2 is invoked by the call to `name-fix` at the end of Algorithm 1.

Algorithm 1: Pseudocode for the Function Inlining Algorithm

Input: fun-call-node, parsed AST_S

Data: ls-static-semantics-specification
Data: language-specific strategies

```
// Step 0 - Prepare id's for name-fix.
foreach s ∈ AST_S do
    if is-string(s) then s.ID := unique-id;
end

// Step 1 - Find Definition and Extract Data.
name-resolution_S := ls-static-semantics-specification(AST_S);
(fun-call-name, args) := ls-extract-data(fun-call-node);
fun-def-name := name-resolution_S(fun-call-name);
fun-def-node := ls-extract-data(AST_S, fun-def-name);
(params, body, ftype) := ls-extract-data(fun-def-node);

// Step 2 - Instantiate Parameters.
// The IDs of params should be kept too.
// Only the declaring identifiers should be marked with "inlined".
params := annotate-nodes(params, "inlined");
param-decs := ls-make-decs(params, args);

// Step 3 - Deal with Return Statements.
(body, return-expression) := ls-fix-control-flow(body, ftype);
return-expression := annotate-nodes(return-expression, "inlined");
foreach s ∈ body do
    if ls-is-return(s) then abort;
end
body := annotate-nodes(body, "inlined");

// Step 4 - Transform the AST.
block := ls-construct-block(param-decs, body);
if ls-is-expression-language then
    AST_T := replace-node(AST_S, fun-call-node, block);
else
    fun-call-stmt := ls-enclosing-statement(AST_S, fun-call-node);
    AST_T := ls-insert-before(AST_S, fun-call-stmt, block);
    if return-expression ≠ Null then
        AST_T := replace-node(AST_T, fun-call-node, return-expression);
    else
        // Void functions don't have a return-expression.
        AST_T := ls-remove-expression(AST_T, fun-call-node);
    end
end

// Step 5 - Check and fix reference relations.
AST_T := name-fix(AST_S, AST_T, name-resolution_S);
return layout-preserving-unparse(AST_S, AST_T);
```

Algorithm 2: Pseudocode for the name-fix algorithm adjusted for inline-function

```
Input: ls-static-semantics-specification
Input: AST_S, AST_T, name-resolution_S

// Try to fix Static Semantics Violations.
while true do
    try:
        name-resolution_T := ls-static-semantics-specification(AST_T);
        break;
    catch error:
        AST_T := ls-fix-semantic-error(AST_T, error);
        // ls-fix-semantic-error should return Null if no error is fixed.
        if AST_T = Null or AST_T is unchanged then abort;
    end
end

// Construct name graphs.
(V_S, ρ_S) := get-name-graph(AST_S, name-resolution_S);
(V_T, ρ_T) := get-name-graph(AST_T, name-resolution_T);

// Rename inlined declarations that are referenced by non-inlined nodes.
// Restart name-fix if something changed.
AST_T := fix-inlined-declarations(AST_T, ρ_S, name-resolution_T);
if AST_T changed then return name-fix(AST_S, AST_T, name-resolution_S);

// Find reference relations that indicate capture.
captured-relations;
foreach v ∈ domain(ρ_T) do
    if v ∈ domain(ρ_S) then
        if ρ_S(v) ≠ ρ_T(v) then captured-relations(v) := ρ_T(v);
    else
        if v ≠ ρ_T(v) then captured-relations(v) := ρ_T(v);
    end
end
if captured-relations = ∅ then return AST_T;

// Fix capture by renaming.
foreach v_d ∈ codomain(captured-relations) do
    new-name := unique-name();
    X_d := get-nodes-from-id(AST_T, v_d);
    foreach x_d ∈ X_d do
        AST_T := AST_T.rename(x_d, new-name)
    end
    foreach v_r ∈ domain(ρ_S) do
        if ρ_S(v_r) = v_d then
            X_r := get-nodes-from-id(AST_T, v_r);
            foreach x_r ∈ X_r do
                AST_T := AST_T.rename(x_r, new-name)
            end
        end
    end
end
return name-fix(AST_S, AST'_T, name-resolution_S)
```

Chapter 4

Implementation

This chapter details our implementation of the Inline Function algorithm described in Chapter 3. It starts by introducing the language workbench Spoofax, including the tools that are used by the implementation in Section 4.1. Section 4.2 then gives an overview of the implementation in Stratego supported by code snippets. Section 4.3 extends the implementation with Statix's multi-file analysis. The language-specific strategies required for the implementation are listed with a brief description in Section 4.4. The chapter concludes by explaining how the implementation can be deployed and used in Spoofax in Section 4.5.

At the time of writing the code has not been integrated into Spoofax's core yet. Instead, you can find the source code as it is tested for multiple languages in a GitHub repository[1]. For each language, the main code file is found in the "trans" directory and is called "inline-function-source.str". The code related to reference relations can be found in "inline-function-name-fix.str" and the language-specific functions can be found in "inline-function-<language>.str".

4.1 The language workbench: Spoofax

Spoofax is a language workbench, a set of tools to aid in the design, implementation and extension of languages [6]. Its main use case is the development of Domain Specific Languages [7, 8]. Developing a language is difficult [22]. First, the language needs a complete grammar. Then, a syntax parser that follows the grammar is required. Often there is a need for static semantics analysis of parsed programs, for example to recognize type errors before compilation. Then program transformations may need to be defined. Function inlining is an example of such a transformation. Finally, the parsed-, analyzed- and possibly transformed syntax needs to be translated back into source code (compilation). Spoofax isolates each of these steps into an individual component of language development. For each component, a separate tool is included that is tailored to its specific needs. But maybe even more importantly, the whole framework is integrated into the development environment Eclipse, making development much more accessible and even providing IDE support for the languages that are developed using Spoofax.

For our inline function refactoring all these components of Spoofax are needed (except for a compiler). To get an AST, the syntax needs to be parsed. To get the reference relations, the AST needs to be analyzed. Inlining the function call requires transforming the AST. And finally, integration with Eclipse offers a user-friendly IDE button to invoke the refactoring. Each component of Spoofax that is used by our implementation is introduced below. An overview of the pipeline of these components in Spoofax can be seen in Figure 4.1.

[1] https://github.com/MetaBorgCube/function-inlining

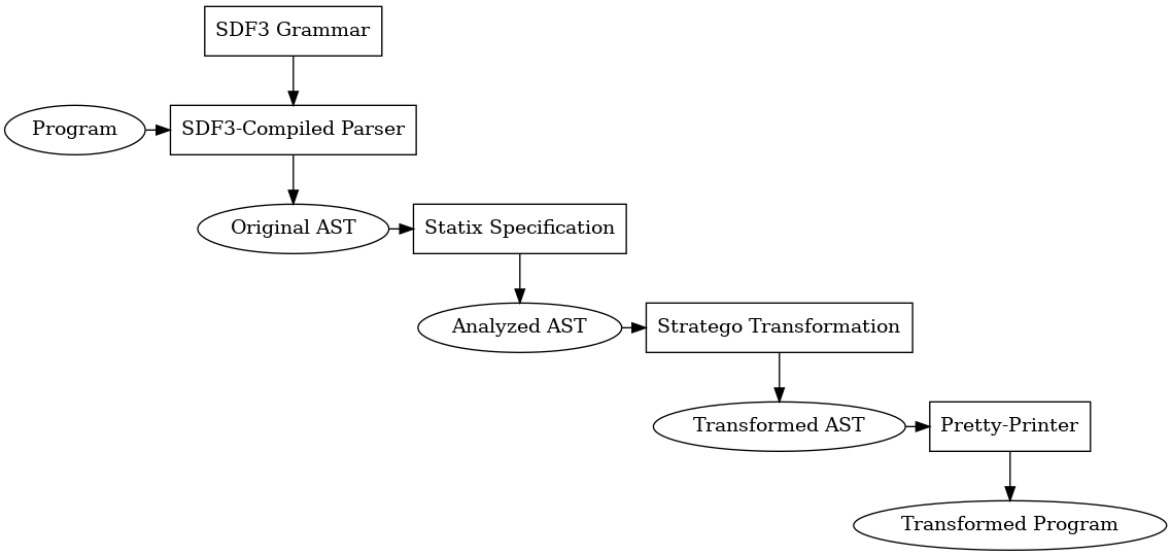


Figure 4.1: The pipeline of applying a Spoofax Stratego transformation to a program.

The Syntax Parser (SDF3). A language is defined by its grammar and semantics. The grammar formalizes the syntax of the language. SDF3 is a language in which context-free grammars can be defined and extended using extra features. The extra features include layout constraints, modular composition and the definition of disambiguation rules [9].

Grammars defined in SDF3 consist of productions. A single production defines a possible structure for a non-terminal. For example, `Statement.Return = <return <Exp>>` is a production for the `Return` alternative of the non-terminal `Statement`. Productions define both the lexical- and the context-free syntax of the language. There are some useful features in SDF3 that make it easier to define productions, such as regular expressions for terminals, but we will not go into detail here since the development of a grammar is not in scope of this thesis.

The production rules of a grammar describe valid AST nodes. The tree is abstract because lexical details are not included in the tree (such as the return keyword in the example above). Together with disambiguation rules, SDF3 can compile grammars to generate parsers that parse to ASTs. The labels of non-terminal AST nodes are taken from the productions, so the production of the `Statement` non-terminal mentioned before has the label `Return`. It should be noted that in SDF3 the left-hand side of a production, the non-terminal & constructor combination, needs to be unique.

Unparsing. Unparsing ASTs back to source code is also incorporated in Spoofax. The SDF3 compiler calls this formatting and makes a distinction between ”ugly-” and ”pretty-printing”. Ugly-printing translates an AST to a program string and inserts some notational details that were abstracted away, such as keywords like ’if’ or ’return’. Ugly-printing only requires the original grammar. Pretty-printing on the other hand adds whitespace to the program to improve readability. This cannot be done from the grammar alone but requires the productions to specify their desired formatting. SDF3 allows users to define the whitespace layout in template productions [23]. From the grammar and the extra template productions, SDF3 generates an unparser.

This unparser does not maintain the original whitespace layout of a program (including comments). If a program has a whitespace layout that does not follow the standard layout from the grammar productions, converting the AST back to a program loses the original whitespace layout. As mentioned in Section 3.1.2 there exists a layout-preserving unparsing algorithm [18] that relies on origin tracking [19]. This algorithm is actually implemented in

Spoofax as the strategy `construct-textual-change`.[2]

Static Semantics (Statix). The Static Semantics of a language can be defined using Statix [11]. Its main purpose is to define type systems of languages with type-checking constraints. In Statix, this is achieved by defining constraints over AST nodes. When SDF3 compiles a grammar it generates algebraic signatures for Statix. Statix uses these signatures to access the sorts and constructors of the grammar in the form of terms.

A key concept of Statix is scope graphs [24, 25] because type-checking requires name resolution. A scope graph represents the naming structure of a program. It represents name bindings and can be queried for name resolution. This is the main reason that Statix is useful for our inline function algorithm: we query the scope graph generated by Statix to determine reference relations. To be able to uniquely reference terms from the scope graph, Statix adds "TermIndex" annotations to all AST nodes. A TermIndex is an ATerm constructor containing the filename as a string and a unique integer index.

Transformations (Stratego). In Spoofax programs are transformed by altering the AST of the program. Automatic AST transformations are defined in the language Stratego [13]. Stratego provides many built-in strategies that ease the traversal- and thereby the transformation of an AST. Stratego transformations are defined by *strategies* that apply *rewrite rules* to nodes in the AST. The signatures generated by the SDF3 compiler are also used by Stratego to access the sorts and constructors of the grammar that form the AST. This way nodes in the ASTs of a language can be interpreted without requiring a redefinition of the productions from the grammar. Note that Stratego considers nodes of an AST to be a *Term*, and its child terms are subterms.

A rewrite rule has a left- and right-hand side term pattern (L and R), where L tries to pattern-match the input term and R represents the term that replaces the input term. When applied to a term, the term is matched to pattern L. In case it matches, variables in L are instantiated with the corresponding input values. For example, when pattern `FunCall(n, a)` is applied to `FunCall("foo", [2])`, we get `n := "foo"` and `a := [2]` in the context of the rewrite rule.

A rewrite rule can also have a 'where' or 'with' condition. This condition consists of a sequence of strategy expressions. A common use case of the condition is to fill in variables of the right-hand side R of the rewrite rule. For example, if R is `Int(a)` and the 'where' condition contains the assignment `a := 2`, the rewrite rule returns `Int(2)`. The condition is also commonly used to invoke other strategies that transform subterms, check certain properties or generate new terms.

Stratego also has access to a Statix API. The API contains a strategy that can run the Statix analyzer on an AST, after which it can use other API strategies to query the analysis result. The most important API call for function inlining is `stx-get-ast-ref` which receives a term with a TermIndex, queries the scope graph and returns a referenced declaration TermIndex, if there is any.

Our inline function algorithm is implemented in Stratego. It is accessible as a Stratego module. When a language engineer wants to use inline function in their language, they first have to implement the required language-specific strategies in Stratego. Stratego allows strategies to be passed along as parameters. To invoke inline function, the main inline function strategy has to be called with the necessary language-specific strategies and program data as arguments. The strategy returns the transformed program in which the function call has been inlined, or it displays an error in case something went wrong.

[2] Unfortunately, at the time of writing there is a bug in the origin tracking implementation in Spoofax, related to retrieving the parent of a list ATerm. Therefore, our implementation cannot use the layout-preserving unparser and uses the 'regular' SDF3 unparser instead.

Editor Services (ESV). Spoofax automatically generates Eclipse editor plugins for defined languages. This includes automatic services such as syntax checking and type checking, but also custom features like syntax highlighting. The custom editor services are configured in ESV files (Editor SerVice). Among other things, ESV files can bind an Eclipse menu button to a Stratego strategy. Clicking an action menu button will invoke the Stratego strategy with predefined arguments that include an AST of the program and a selected term. The strategy returns a program, which is then printed to a specified file. This is useful for our inline function implementation because it allows us to define an Eclipse button that invokes the inline function strategy.

4.2 The Inline Function Strategy

The Inline Function algorithm as described in Algorithm 1 is implemented as a Stratego rewrite-rule called `inline-function-action` (Figure 4.2). It receives all language-specific strategies as strategy arguments. Furthermore, its left-hand side is a 5-tuple including the selected function call term, the parsed- and analyzed AST and the path to the source code file. With its right-hand side it returns a tuple containing the filename of the program and the transformed source code, or `None()` if an error occurred. The actual algorithm is implemented in `inline-function`, `inline-function-action` is needed to handle some details of the ESV framework.

```
1  inline-function-action(language-specific-strategies | extension) (selected-term,
       _, ast, path, _) -> (filename, transformed-program)
2    where
3      ...
4      transformed-ast := <inline-function(...)> (selected-term, ast)
5      ...
6
7  inline-function(language-specific-strategies | error-container) (selected-term,
       ast) -> transformed-ast
8    where
9      ...
```

Figure 4.2: The main Inline Function Strategy header.

When the strategy is invoked the program is already parsed and analyzed. At the start of the implementation the analysis object is extracted by calling `stx-get-ast-analysis` on the AST. This object can be used to query reference relations of the original program. Next, the implementation starts performing the steps from Algorithm 1. The sections below describe the Stratego implementation of each step in more detail, starting with the generation of unique IDs for identifiers in Section 4.2.1.

4.2.1 Annotating Terms

The first step in Algorithm 1 is assigning unique IDs to the identifiers in the AST. So, we need to visit every identifier term in the AST. Identifier terms can be recognized by the fact that they consist of only a string value (that represents the identifier). The problem is that if two different terms hold the same string value — for example because a variable is used twice throughout the program — there is no way to distinctively reference them when binding unique IDs. Now, in an analyzed AST in Spoofax, every term receives a unique `TermIndex` annotation. You might think that using the term indexes as IDs is a good solution, but unfortunately when an AST changes and is re-analyzed the term indexes are recreated, so there

is no guarantee that the already existing terms will keep the same term indexes after the transformation. Additionally, the terms that are copied for function inlining should keep the same ID as their original counterpart, but would receive a different term index when the AST is re-analyzed. Therefore, we opt to add an annotation to each term that holds its ID. When a term is copied, the annotations are preserved. The same holds for re-analysis by Statix, custom annotations are not removed.

The process of generating and annotating the IDs demonstrates a useful Stratego strategy for traversing ASTs: `topdown(s)`. It traverses the AST from left to right in a depth-first manner starting from the root term. It applies strategy `s` to every term it visits, consequently applying `s` to every term in the AST. Strategy `s` itself can consist of a sequence of strategy expressions, so we can use it to generate the unique ID and add it to the set of IDs. As a result, generating and annotating the unique IDs is performed by the code snippet shown in Figure 4.3.

```
1     vs := <new-iset>
2   ; new-ids := <new-hashtable>
3   ; (ast-s, asts-s) := <topdown(try(
4                  is-string;
5                  generate-id(|vs, new-ids, NameFixID(<local-newname> "v"))
6                 ))> (ast, asts)
```

Figure 4.3: The Stratego expressions that generate unique IDs in `vs` and annotate the identifiers with the IDs.

4.2.2 Extract Data

After generating unique IDs the required information needs to be extracted from the AST. First, we need the identifier of the function call in order to get the referenced function definition. This can be combined with extracting the arguments from the function call. As explained in Section 3.3 we assume that a language-specific strategy `ls-extract-data` is available that pattern matches a function call term and returns the identifier- and argument terms. Now, there is a possibility that when selecting a function call in the source code, the corresponding selected term from the AST has a constructor wrapper around the Call constructor. During our testing phase this occurred with a C++ grammar when the function call is the only expression in a statement, in which case it is wrapped by an Exp constructor. To solve this issue we traverse the selected term in a topdown manner until the language-specific strategy recognizes a function call.

Now that we have extracted the function identifier we can use it to query the referenced function definition identifier in the scope graph. We use the Statix API strategy `stx-get-ast-ref(|analysis)` on an identifier term to get the referenced term (here `analysis` represents the Statix analysis result). Note that if a list of referenced identifiers is returned, we only continue if the list contains exactly one identifier term. Otherwise, it is unknown which exact identifier will be referenced (due to dynamic binding) so we have to abort.

To get the full function definition term from its identifier term we need another language-specific rule `ls-extract-data(|fun-def-identifier)` that matches a term with a function definition constructor containing our identifier term. An example of such a rule for a simple function definition constructor can be seen in Figure 4.4.

The issue now is that the Statix scope graph has an AST without name-fix ID annotations, so the returned identifier term is not annotated with a `NameFixID(v)`. Because of this the language-specific strategy cannot match the returned term with the annotated term in the AST. To fix this we create the hash table `new-ids` in Figure 4.3 which maps the unannotated terms to their annotated variants. This way we do not have to completely re-analyze the

```
1   ls-extract-data(|fun-def-name): FunDef(fun-def-name, ftype, params, body) -> (
        params, body, ftype)
```

Figure 4.4: The language-specific rule that matches the function definition from its identifier term.

annotated AST just so that the scope graph returns annotated terms, which would be highly inefficient.

4.2.3 Creating Parameter Declarations

In the previous step we extracted the parameter declarations and arguments. To turn them into parameter declarations we need to visit them one by one and fill in a variable definition term. However, since the structure of a variable definition term depends on the grammar of the language in question, we need a language-specific strategy that can turn them into declarations. The returned declarations are only required when combining them with the body, which is done by another language-specific strategy. So, whether the resulting declarations are returned as a list or as some Term in the language in question is up to the language engineer that writes both language-specific strategies.

It is important that we annotate the declaring identifiers with `"inlined"` for the name-fix process. However, we should not annotate the arguments with `"inlined"` because they are already present in the function call scope (see Section 3.7.4 for more information). Therefore, we need to add this annotation to the parameters only before combining them with the arguments.

4.2.4 Removing the Return Statement

Section 3.5 explained that for some languages, but not all, the last statement is a return statement that should be removed to maintain the control flow of the function body. Because this step differs per language, it relies mainly on the language-specific strategy `ls-fix-control-flow`. The strategy is supposed to strip the last return statement from the body and return the expression it contains. Now, this strategy does not have to be complex. For expression languages in which the function body is just one big expression there is no last return statement, so the strategy can just return the original body and nothing for the return-expression (we use the ATerm '`()`' to represent 'nothing'). For statement languages, if the function type is void, the return-expression should be '`()`' as well, and the last statement should only be removed if it is a return statement.

As argued in Section 3.5 we do not handle early return statements generically at the time of writing, but we do allow the language engineer to handle them in `ls-fix-control-flow`. Therefore, we have to check that there are no return statements left after removing the last return statement. This can be done with a simple `topdown` traversal through the function body with a language-specific pattern match on return statement terms. If remaining return statements are found, we abort and print an error.

4.2.5 Inserting the Code Block

With the return statements dealt with, it is time to transform the AST. First, the parameter declarations and function body have to be combined into a single code block. Concatenating the parameter declarations and the body requires language-specific knowledge of the structure of such a sequence. It is performed by `ls-construct-block`. Note that the `"inlined"` annotation has been added to the parameter declarations before but not to the body terms, so we have to do that before combining them into a single code block.

Then the resulting block needs to be inserted. This process differs a lot per language. First, the strategy `ls-is-exp-language` indicates whether we are dealing with an expression language or a statement language. It should just be implemented as `id` for expression languages and `fail` for statement languages. This way, we can wrap it in a Stratego `if` condition and separate the processes for the two types of languages.

Inserting the block for expression languages only involves replacing the function call term with the block, assuming that `ls-construct-block` yields a correct expression. What this looks like in Stratego can be seen in Figure 4.5. The strategy `oncetd(s)` traverses the AST like `topdown(s)` but once the strategy `s` succeeds for a single term, it aborts. The expression `?fun-call-term` tries to pattern match on the current term and `!block` replaces the current term by `block`.

```
1  ; block := <ls-construct-block> (<topdown(add-anno("inlined"))> body, param-decs)
2  ; if ls-is-exp-language
3    then ast-t := <oncetd(?fun-call-term; !block)> ast-s
4    else ...
5    end
```

Figure 4.5: The insertion of the code block in Stratego for expression languages.

The else clause handles statement languages and is more involved. It starts by finding the statement term surrounding the function call expression. To achieve this, a language-specific strategy `ls-is-statement` is needed that succeeds when applied to any statement term and fails otherwise. Building a language in Spoofax generates Stratego strategies called injections. Injections are mainly generated for Statix and Stratego to access the grammar. The injections should include one that can determine whether a term is a statement or not. This is exactly what we need for `ls-is-statement`.

With this strategy we can find the surrounding statement. We defined a recursive strategy that starts at the root node of the AST, and recursively calls itself on each subterm. If the subterm is a statement term, it is stored as the last-encountered statement. If the visited term is the function call expression, we return the last-encountered statement as the surrounding statement. Else, we recursively call the strategy on each subterm of the current term.

The strategy `one(s)` comes in handy here. It tries to apply `s` to each subterm and stops when one succeeds. Our implementation of finding the surrounding statement can be seen in Figure 4.6. Note that if you specify multiple definitions of a rewrite rule, Stratego tries to apply them one by one until one succeeds. We highlighted the main differences between each rule in blue in an effort to make it more apparent what their purpose is. For example, the third rule checks if the current term is a statement, in which case it stores it as the last-encountered statement.

Once the surrounding statement is found, the constructed block needs to be inserted right before it. Unfortunately this cannot be performed generically as explained in Section 3.6. Instead, we invoke the strategy `ls-insert-before` with the discovered statement, constructed block and AST as arguments. It is expected to insert the constructed block in the sequence of statements that contains the discovered statement. Language-specific special cases like when the surrounding statement is a for-loop condition are expected to be either rejected or correctly implemented by `ls-insert-before`.

The last task that remains for statement languages is replacing/removing the function call. There is a distinction here between void functions and other functions. Void function calls need to be removed, whereas the other function calls need to be replaced by the return expression. Luckily we have `return-exp = ()` for void functions so we can separate the two cases with an if-check on the contents of return-exp. If it equals `()` we need to invoke a language-specific strategy `ls-remove-exp` that correctly replaces the call with a valid empty

```
1  get-surr-stmt(ls-is-stmt|fun-call-term, _): _ -> fun-call-term
2    where
3      <ls-is-stmt> fun-call-term
4  get-surr-stmt(ls-is-stmt|fun-call-term, last-stmt): fun-call-term -> last-stmt
5  get-surr-stmt(ls-is-stmt|fun-call-term, last-stmt): t -> result
6    where
7        <ls-is-stmt> t
8      ; <one(get-surr-stmt(ls-is-stmt|fun-call-term, t); ?result)> t
9  get-surr-stmt(ls-is-stmt|fun-call-term, last-stmt): t -> result
10     where
11       <one(get-surr-stmt(ls-is-stmt|fun-call-term, last-stmt); ?result)> t
```

Figure 4.6: Finding the surrounding statement.

'void' expression. Otherwise, we can just generically replace the function call expression with the return expression as displayed in Figure 4.7.

```
1  if <eq> (return-exp, ())
2  then ast-t' := <ls-remove-exp(|fun-call-term)> ast-t
3  else ast-t' := <oncetd(?fun-call-term; !return-exp)> ast-t
4  end
```

Figure 4.7: Replace/remove the function call expression.

4.2.6 Applying name-fix

Analyzing the Transformed AST

In Stratego, the Statix scope graph of a program has already been constructed when an ESV menu action is invoked. It can be retrieved by calling `stx-get-ast-analysis` on any term of the AST. When an AST is annotated or transformed this cached scope graph is not updated. To update the scope graph the Statix analysis has to analyze the new AST. However, where at first the Statix analysis was automatically applied in the background, this time we need to manually invoke the analysis. This requires a specific input wrapper that contains among others the AST, the old scope graph and the path to the program. The constructor names of this wrapper differ for single-file and multi-file analysis, more on that in Section 4.3. Applying re-analysis returns another wrapper containing the new scope graph, the new AST with updated `TermIndex` annotations (the other annotations such as `NameFixID` are preserved) and a list of semantic errors that were found.

To analyze the AST and fix potential semantic errors we implemented the recursive strategy `fix-semantic-errors`. It fixes errors and then recursively calls itself until no errors are left. As already explained in Section 3.7, the semantic errors are fixed by exposing them to a language-specific strategy `ls-fix-semantic-errors`. Statix error messages are custom strings so there is no way we can interpret their meaning generically. Instead, the engineer of each language can decide to fix- or reject them in the language-specific strategy. If a discovered error remains unfixed, the inline action is aborted and the original program is restored. If `ls-fix-semantic-errors` claims to have fixed an error but did not change the AST, we abort to avoid getting stuck in an infinite loop.

Setup for Name-Fix

When the new AST is analyzed and there are no static semantics errors left, we need to construct the name graphs for name-fix. Section 3.7 defines a name graph as a tuple (V, ρ) where V is the set of identifier IDs and ρ maps the ID of each referencing identifier to the ID of the referenced declaration. At the start of our transformation we already constructed V while generating the IDs. Now, name-fix keeps the same ID for copied terms, and inline function only inserts copied terms. So, the set of IDs V after transforming the AST only differs in the fact that the ID v_c of the function call identifier is removed. Because v_c is not in the new AST there is no way for it to be in the domain of ρ_T and thus it cannot interfere with name-fix. Therefore, we opted to not bother removing v_c from V_T and instead reuse V_S for V_T.

Constructing ρ is simple. Visit every AST term where `stx-get-ast-ref` returns a declaration and add the two NameFixID annotations as a relation to ρ. For the original AST a declaration returned by `stx-get-ast-ref` does not have the NameFixID annotation, but the hash table `new-ids` maps the term to a term with the correct annotation. Our Stratego strategy that constructs ρ can be seen in Figure 4.8, where `analysis` is the scope-graph.

```
1   rho := <mapconcat(construct-ref-relations(|new-ids))> [(ast, analysis)|asts-
        analyses]
2
3   construct-ref-relations(|new-ids): (ast, analysis)
4     -> <collect(get-ref-relation(|analysis, new-ids))> ast
5
6   get-ref-relation(|analysis, new-ids): t -> (v-r, v-d)
7     where
8         is-string
9       ; v-r := <get-name-fix-id> t
10      ; t-d := <stx-get-ast-ref(|analysis)> t
11      ; (v-d := <get-name-fix-id> t-d
12      <+ v-d := <get-name-fix-id> <hashtable-get(|t-d)> new-ids)
13
14  get-name-fix-id: t -> v
15    where
16      NameFixID(v) := <fetch-elem(?NameFixID(_))> <get-annotations> t
```

Figure 4.8: The strategy that constructs ρ.

Checking for Capture

With the name graphs completed, checking for capture is not very difficult. Getting the domain of ρ is just a simple `map(Fst)` call since it is a map of tuples representing reference relations. Checking for capture follows Definition 1 from Section 3.7: Loop through the domain of ρ_T and check each of the definition's properties. Note that property 2 is not checked as every ID v will be in V_S, as explained at the end of Section 3.7. Every time a property is violated, the corresponding tuple from ρ_T is stored as a capturing relation. The implementation in Stratego can be seen in Figure 4.9, where all capturing relations are collected in `cap-rel`.

Fixing by Renaming

If the list with capturing relations is empty, we are finished with name-fix. Otherwise, we have to rename identifiers to try to fix capture. In the algorithm we loop through the codomain of the captured relations, i.e. the declarations causing capture. Getting the co-

```
1  cap-rel := <collect(check-capture(|dom-rho-s, rho-s, rho-t))> dom-rho-t
2
3  check-capture(|dom-rho-s, rho-s, rho-t): v -> (v, v-d)
4    where
5      if <elem> (v, dom-rho-s)
6      then v-d' := <rho-lookup(|v)> rho-s
7        ; v-d := <collect-one(where(Fst; ?v); Snd; not(?v-d'))> rho-t
8      else v-d := <collect-one(where(Fst; ?v); Snd; not(?v))> rho-t
9
10 rho-lookup(|v-r) rho
11   -> <Snd> <fetch-elem(where(Fst; ?v-r))> rho
```

Figure 4.9: The strategy that collects all capture relations in `cap-rel`.

domain is simple: `map(Snd)`. However, this can yield duplicates, so we turn the list into a set with `make-set` to avoid redundantly renaming multiple times.

We need to loop through the co-domain and rename all terms with those IDs, together with all terms that referenced those IDs in the original AST. In our implementation we show off Stratego's built in `foldr(!v, s)` strategy, which applies `s` to the first element of the list and `v`, then it applies `s` to the result of that and the second element of the list, etc. until the list is traversed and there is a single remaining value (in our case the AST with renamed terms). Figure 4.10 shows what the renaming process looks like in Stratego code.

```
1    codomain-cap-rel := <make-set> <map(Snd)> cap-rel
2  ; ast-t-renamed := <foldr(!ast-t, rename-capture(|rho))> codomain-cap-rel
3
4  rename-capture(|rho-s): (v-d, ast) -> new-ast
5    where
6        x-d := <collect-one(where(get-name-id; ?v-d))> ast
7      ; new-name := <local-newname> x-d
8      // Rename declarations with ID v-d
9      ; ast' := <topdown(try(has-id(|v-d);
10                            preserve-annos(!new-name)))> ast
11     // Rename terms that reference ID v-d
12     ; new-ast := <topdown(try(references-id(|v-d, rho-s);
13                               preserve-annos(!new-name)))> ast'
14
15 has-id(|v): t -> t
16   where
17     <eq> (v, <get-name-fix-id> t)
18
19 references-id(|v-d, rho): t -> t
20   where
21       v-r := <get-name-fix-id> t
22     ; <eq> (v-d, <rho-lookup(|v-r)> rho)
```

Figure 4.10: The strategies that rename terms based on the capture relations.

After renaming terms inside the AST, we have to recursively call name-fix until there is no capture left. This either fixes all capture cases meaning our inline function transformation is finished, or it introduces new semantic errors which cause us to reject the inline action

(for example because an inlined reference was captured, but after renaming the capturing declaration it turns out that the intended declaration is unreachable, and a reference that cannot reference a declaration is a static semantics error).

4.2.7 Recursive Function Calls

Section 3.7.4 explains that name-fix could miss capture when inlining *recursive* function calls. It states that an inserted declaration could shadow the original declaration that it is copied from, meaning that it has the same NameFixID as the declaration it shadows. The mentioned solution is to annotate all inserted terms with `"inlined"` such that we can recognize them. Then, we have to check that no identifier *without* the annotation references a declaration *with* the annotation. We already added the annotations in the previous steps.

Checking the transformed AST to see if a non-inlined term references an inlined declaration is simple now. We explore all string terms in the AST without the `"inlined"` annotation. If one of them references a term with the annotation, it means the inserted declaration is capturing and its NameFixID is collected for renaming.

We now have a list with the IDs of all capturing declarations, similar to when name-fix starts renaming. We also have ρ_S, showing which IDs are intended to reference the collected IDs. Name-fix would rename all IDs with no exception. Here, we do the same, except we only rename terms annotated with `"inlined"`. That way the original terms from the function definition remain unchanged.

4.2.8 Fixing Duplicate Declarations

Section 4.2.6 introduced the strategy `ls-fix-semantic-errors` which exposes static semantics errors for the language engineer to fix them. Duplicate declarations fall under these errors and are therefore expected to be fixed by the language-specific strategy. As Section 3.7.5 mentions we implement a helper strategy that fixes duplicate declarations because it can be done generically.

The helper strategy receives the duplicate declaration term, and the original- and transformed ASTs with the NameFixID- and `"inlined"` annotations. First, the NameFixID annotation v_d of the duplicate declaration is extracted. The ID is used to find the corresponding term in the original AST. Then, we collect the NameFixIDs of all terms that reference this term in the original AST. A new name is chosen using the built-in `local-newname` strategy and all terms with the collected IDs that also have the `"inlined"` annotation are renamed, ending with the duplicate declaration term itself.

4.3 Multi-File Inline Function

Spoofax distinguishes between *single-file* and *multi-file* mode. A language implementation is set to one of these two modes. In single-file mode Spoofax analyzes program files in isolation, whereas in multi-file mode all files in a project are analyzed simultaneously such that, for example, references can span multiple files. As you can imagine this mainly concerns the Statix analysis.

What this means for us is that the function definition may be located in a different file, so to extract the necessary information we need to explore all ASTs in the project. Additionally, the re-analysis strategy that applies the Statix analysis to the transformed AST has to be invoked differently. After all, analyzing in single-file mode only requires the AST of the file with the function call whereas analyzing in multi-file mode requires the ASTs of all files in the project. Finally, since there can be references from one file to another in multi-file mode, name-fix will need to explore references from the other files as well when constructing the name graphs.

It should be clear by now that our implementation needs to distinguish between single-file and multi-file mode. To include both modes we require a language parameter that indicates which of the two modes is used in the language implementation. Based on this parameter our implementation chooses the input for the re-analysis strategy, and chooses the ASTs to explore when constructing name graphs for name-fix.

4.3.1 Finding the Function Definition in Multi-File Mode

In multi-file mode, the Statix analysis corresponding to the AST of a single file supports references to ASTs of other files. So if the function call references a function definition in another file, we can still query that reference in the analysis of the AST containing the function call.

As we explained before, the result of a reference query is not the full function definition term but only the subterm holding the function name. To extract the remaining terms of the function definition (body, parameters, etc.) we need to explore the AST containing the function definition. In multi-file mode, this can be any AST in the project. Luckily the Stratego strategy that we employ in single-file mode to traverse the AST (`collect-one`) can traverse ASTs *and* lists. So, by combining the selected AST and all other ASTs in a single list, we can extract the function definition term the exact same way as in single-file mode. The only difference is that in single-file mode the list of other ASTs (`asts-s`) will be empty:

```
(params, body, ftype) := <collect-one(ls-extract-data(|fun-def-name))> [ast-s|asts-s]
```

4.3.2 Analyzing in Multi-File Mode

A changed AST can be re-analyzed with a call to `stx-editor-analyze`. This API function receives an `AnalyzeSingle` or `AnalyzeMulti` term, depending on the selected mode. For single-file mode, this term only contains the transformed AST and its previous analysis object. Multi-file mode requires all ASTs and their previous analysis objects, together with the analysis object of the full project. The strategy returns the updated ASTs and analyses.

Now, a language implementation in Spoofax creates an Eclipse editor for said language. Program files of that language are parsed and analyzed by the editor. This yields analysis results for all files that are cached by Spoofax. In Stratego these results can be extracted at any time with Statix API calls. However, when an AST is transformed and re-analyzed with an `editor-analyze` call, the cached analysis results are not updated. The updated analysis results are only present in the return value of `editor-analyze`. So, after the first re-analysis we cannot use the Statix API call to retrieve the analysis anywhere we want, since it will return an outdated analysis object. Instead, we need to pass around the analysis results after each re-analysis to be able to use them.

4.3.3 Name Graphs in Multi-File Mode

In single-file mode we construct the name graph by exploring the AST of the selected program. For every term with a NameFixID we query the Statix analysis to determine a reference relation for the name graph. In multi-file mode we need to do the same thing, but for all ASTs and then combine all relations into a single name graph.

To do this, we keep track of the ASTs and their analyses in a list of AST-analysis tuples. At the start of our implementation where we add NameFixID annotations to the selected AST, we give the other ASTs unique NameFixID annotations as well. Then before invoking the name-fix strategy we combine those ASTs with their analyses in said list of tuples. Just like the way we find the function definition term (Section 4.3.1) we combine the AST with the function call and the other ASTs in a list such that we do not have to create two different implementations for single-file and multi-file mode (note that `ast-s` is the selected AST, `stx-s` is its analysis and `asts-stxs` is the list of other AST & analysis tuples):

```
rho-s := <mapconcat(construct-ref-relations)> [(ast-s, stx-s)|asts-stxs]
```

4.3.4 Multi-File Exception

It is important to note that we do not actually fix capture when it takes place in other files than the file containing the function call. Here is a situation in which this happens: suppose that we are inlining a function call to `foo` in a file called `main` that imports the files A and B in that order. Both A and B contain a global variable declaration x. The definition of `foo` is located in B and contains a reference to x in B. Now, the static semantics of the language in question determine that since A is imported first, referencing x in `main` would point to the x in A. This means that inlining `foo` in `main` causes the reference to x in B to be captured by x in A. Although our implementation of name-fix will detect the capturing declaration, we do not rename it because the declaration is in a different file.

There is a good reason for not renaming other files: The contents of imported files may be out of our control. For example, they can be part of a (public) library. The library code could be referenced by other files that are not visible when refactoring. And even if the other file is exclusively part of your project it can still be written by someone else, so it might be a bad idea to change names in there. If the author of the other file does not realize why you changed the names, they could undo the change without you noticing, reintroducing the capture. Therefore, instead of changing the other files, we recognize the naming conflict and report it to the user instead.

4.4 Overview of Language-Specific Strategies

The set of language-specific strategies is the third language parameter to our implementation. Even though we expect the language engineer to implement them it is still useful to describe what our implementation expects them to do, to get an impression of how much effort it takes to implement them.

In the list below we briefly explain the purpose of each language-specific strategy in the context of our Stratego implementation. We also show an actual Stratego implementation of the language-specific strategies for two languages in the Appendix: C++ (Appendix A) and Tiger (Appendix B). We defined those language-specific strategies as part of the testing chapter (Chapter 5).

- **`ls-extract-data(|n)`**

  This strategy has two purposes. First, it is used to pattern match a function call term and extract the identifier- and argument subterms. When a user selects a program fragment for function inlining, the corresponding AST term is provided to the language parametric implementation. This term contains the function call term, but may have constructors wrapped around it. Our implementation traverses the selected term using `ls-extract-data` to pattern match on a function call term and extract its identifier- and argument subterms.

  Secondly, it is used to pattern match a function definition term with name `n` (`n` is a parameter to the strategy) and extract its parameters, body and type. After using Statix to find the identifier term of the referenced function definition, the generic implementation traverses the AST whilst using `ls-extract-data(|n)` to pattern match on function definitions with identifier term `n`.

- **`ls-make-decs: (params, args) -> decs`**

  `ls-make-decs` receives the parameter subterms (extracted from the function definition term) and the argument subterms (extracted from the function call term). It should

use the variable declaration constructor of the language in question to create terms that initialize the parameters with the argument expressions.

- **`ls-fix-control-flow(|type): body -> (new-body, return-exp)`**

  If the language in question has return statements this language-specific strategy should transform them in '`new-body`' to get rid of the return keywords whilst maintaining the control flow of the function body. It should also extract the return expression that is going to replace the function call expression. In case it only transforms the last return statement of a function body (so not early return statements) our implementation will abort if it detects return keywords in '`new-body`', but it can still be used for function bodies that do not contain early return statements.

  The function type is provided in '`type`' for when a language supports void functions. Void functions can have empty return statements — return statements without return expressions — in which case '`return-exp`' needs to be returned as `()`.

- **`ls-is-return`**

  When called on a term, this strategy should succeed if the term represents a return statement and fail otherwise. In case the language in question does not have return statements this strategy should always fail. It is used by our implementation to check that all return statements have been removed.

- **`ls-construct-block: (new-body, parameter-declarations) -> block`**

  This strategy receives the transformed function body and the list of variable declarations for the parameters. It should use a term constructor of the language in question to combine them into a valid term `block` in which the parameter declarations are executed before the function body. The returned term `block` will later replace/be inserted before the function call term in the AST.

- **`ls-is-exp-language`**

  This strategy should succeed for expression languages and fail for statement languages. Our implementation uses this strategy to determine if it can simply replace the function call expression with the function body (which is possible in an expression language because everything is an expression), or if it should insert the function body before the statement containing the function call (because it is a statement language).

- **`ls-is-statement`**

  When called on a statement term this strategy should succeed, otherwise it should fail. It is essentially the same as the injection that matches statements that should be generated for Statix. This strategy does not apply to expression languages, for which it should always fail.

- **`ls-insert-before(|stmt, block): ast -> transformed-ast`**

  Using `ls-is-statement` the generic code finds the statement '`stmt`' that contains the function call expression. The code constructed by `ls-construct-block` ('`block`') should now be inserted directly before that statement, which depends on the grammar of the language in question. This should yield the `transformed-ast`. For expression languages this strategy is never called so it can just fail.

- **`ls-remove-exp(|exp): ast -> transformed-ast`**

  If a language supports functions with type void, it can be that the return expression is empty. In that case, the language parametric code has to remove the function call instead of simply replacing it with the return expression. However, removing a term

from an AST can create an invalid AST structure, so this language-specific strategy is needed to replace the function call term with the 'void' expression term of the language in question.

- **`ls-fix-semantic-errors(|err, (old-ast, old-analysis), asts-analyses): ast -> fixed-ast`**

  The Statix analysis might discover static semantics errors (`'err'`) in the transformed ast (`'ast'`). These errors are language-specific so they cannot be fixed generically. However, to avoid unnecessarily aborting when an error could easily have been fixed with language-specific knowledge, we expose the semantic errors to the language engineer in this strategy, giving them the option to fix some of the errors. The language developer can invoke our helper strategy `'rename-duplicate-declaration'` here to get rid of a duplicate declaration error.

- **`rename-duplicate-declaration: ((old-ast, old-analysis), asts-analyses, new-ast, duplicate-term) -> fixed-ast`**

  Even though this is not a language-specific function, it is still relevant for `ls-fix-semantic-errors`. The function renames a duplicate declaration and all its intended references to a unique new name. When there is a duplicate definition error there is always a pair of duplicates, one of which is an inserted declaration from the function body. It is important that this inserted declaration is passed along in `duplicate-term` and not the other declaration, otherwise nothing will be renamed.

4.5 Adding Inline Function to a Spoofax Language

To use our implementation in an existing Spoofax language, two things need to be prepared: an Eclipse button that calls the inline function strategy needs to be defined, and the language-specific functions need to be defined in a Stratego file that exposes them to our implementation. Let's start with the Eclipse button.

A Stratego strategy can be bound to an Eclipse menu button with the help of an Editor SerVice (ESV) file. To use our inline function implementation, you should do exactly that. In his Rename refactoring [21], Misteli suggests creating a separate Refactoring ESV file and add "Rename" to it as a menu action. The ESV file is placed in the editor directory of a language implementation. For consistency, we suggest extending the Refactoring ESV file with an inline function menu action. Figure 4.11 shows what this ESV file looks like with inline function added. When the menu action is clicked, the strategy is invoked with the parsed- and analyzed AST, the path to the source code file, the path to the enclosing project and the AST term corresponding to what was selected in the source code.

```
1  module Refactoring
2
3  menus
4    menu: "Refactoring"
5    action: "Rename"= rename-menu-action
6    action: "Inline Function" = inline-function-menu-action
```

Figure 4.11: The ESV syntax for adding a menu action that calls a Stratego strategy.

Next, a Stratego file needs to be created that holds the strategy referenced by the menu action. The file should be placed in the `trans` directory of the language implementation. It should import our implementation and be imported by the main Stratego module of the

language. The language-specific strategies should also be defined in the Stratego file. An overview of the strategies is presented in Section 4.4.

In the same Stratego file the strategy referenced by the menu action should call the strategy `inline-function-action` from our implementation with the language-specific functions in the correct order. To perform SPT tests a separate strategy has to be defined that invokes the strategy `inline-function` instead, because the input and output of SPT tests is different from that of ESV menu actions. An example of the full Stratego file for a C++ project and a Tiger project, including the language-specific strategies, can be found in Appendix A and Appendix B respectively.

4.5.1 Usage

Having added the ESV menu action and the Stratego file that calls our implementation, users of the language in question should now have access to the inline function refactoring. If they highlight a function call in any program file they can click `Spoofax > Refactoring > Inline Function` to invoke our algorithm. See Figure 4.12 and 4.13 for what this looks like in practice.

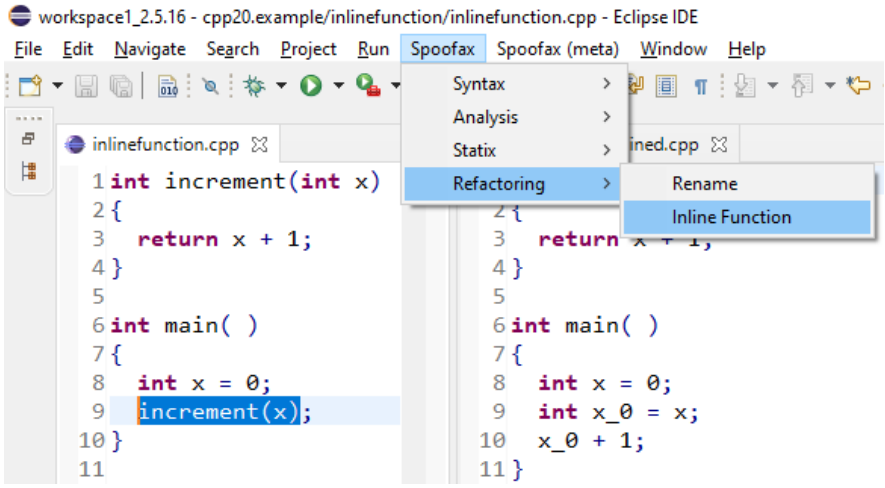


Figure 4.12: Applying the Inline Function refactoring to C++ code in Spoofax.

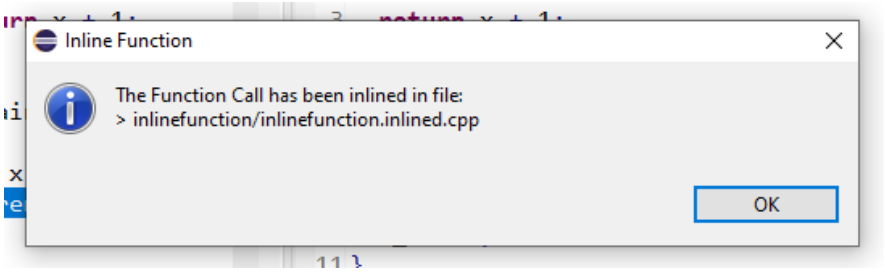


Figure 4.13: The popup message that indicates the refactoring was successfully applied.

If the refactoring was unsuccessful, an Eclipse GUI popup will appear with details as to what went wrong. This could be because the function call in question cannot be inlined or because there are errors in the language-specific functions. The error popup is intended to point out if a language-specific function failed and if so, which one. Next to the name of the function that failed it also shows the input values it received when it failed, to aid in debugging what went wrong.

If nothing goes wrong, the program will be transformed, indicated by the popup message in Figure 4.13. In case the user wants to undo the refactoring the Eclipse undo toolbar button (Ctrl/Command + Z by default) will revert the change.

Chapter 5

Testing

To test the implementation we apply it to various programs that contain function calls. Given the algorithm description in Chapter 3 we can derive an expected output program in which the selected function call has been inlined. We test the implementation by comparing this expected program with the actual output program. The process is automated using SPoofax's Testing language (SPT). The tests cover various edge cases in multiple languages. In this chapter we introduce SPT and give an overview of the test cases that we cover.

5.1 SPoofax Testing (SPT)

The SPoofax Testing language is designed for testing Domain-Specific Languages [14]. SPT is, as the name suggests, part of Spoofax. It integrates the Spoofax components that define a language: the grammar (SDF3), static semantics (Statix), transformation strategies (Stratego) and editor services (ESV). SPT is language-parametric, it can be used by any language defined in Spoofax.

An individual SPT test contains a (fragment of) a program in the language that is being tested. When the test is performed, the fragment is automatically parsed and analyzed. Then, some test conditions are checked as specified by the test. This can be anything from checking if parsing succeeds to comparing transformation results with some expected result. Examples of testing conditions include 'analysis fails' or 'this variable references that declaration'. An SPT test has the following syntactical structure [14]:

```
test name [[
  program
]] condition(s)
```

Components of the program can be exposed to the testing conditions by wrapping them between double square brackets: `[[component]]`. They can then be referenced with `#i` to reference the AST term corresponding to the `i`-th wrapped component of the program.

The testing conditions can access Stratego transformations. So, by wrapping the function call in square brackets we can pass it to our inline function strategy. Invoking a strategy yields a return value. The testing condition can also specify what this value is expected to be, failing the test when the result is different. Applying all of this in practice, the structure of our tests regardless of what language is being tested can be seen in Figure 5.1

The input to a strategy called from an SPT test is the AST representation of the fragment. This is different from the input to an ESV menu action (the IDE button that invokes our refactoring), which receives a tuple of 5 values including the AST. On top of that, the expected output of an SPT test is matched in AST form, whereas a menu action returns a new program string. To deal with this we split our main inline function strategy into `inline-function` (for SPT tests) and `inline-function-action` (for the ESV button). `inline-function` receives the

```
test name [[
  ...
  [[function call]]
  ...
]] run inline-function-test(|#1) to [[
  expected transformed program
]]
```

Figure 5.1: The structure of the SPT tests used to test the implementation.

AST and the selected function call term and returns the transformed AST. `inline-function-action` receives the 5-tuple from the menu action, invokes `inline-function` and then unparses the transformed AST to a program string. The SPT tests invoke `inline-function` directly, skipping the unparsing step.

The SPT language is modular. It provides constructs to specify testing modules that contain a number of tests, and these modules can be executed individually. Having your tests split over multiple modules means that you do not have to re-run all tests after changing one of them. This is very useful for language implementations that take a longer time to parse or analyze. For this reason we grouped our tests in multiple SPT modules. The sections below show test cases we covered, grouped according to the modules we implemented in practice.

5.2 The Test Cases

To be able to run SPT tests for a specific language, its grammar and static semantics have to be defined in Spoofax. Therefore, we are limited to languages with a functioning SDF3 grammar and Statix semantics implementation. Luckily, due to active use of Spoofax in education and research over the past decade, multiple language implementations are readily available. The only thing we have to do before a language can be used for testing is write the language-specific strategies. The languages for which we tested our implementation are listed in Table 5.2.

Table 5.2: An overview of the languages for which we implemented the required language-specific strategies and tested our implementation.

Name	Characteristics	# test cases
C++	General-purpose language, object-oriented, statements, classes, shadowing allowed, duplicate declarations not allowed.	31
WebDSL	DSL, dynamic web applications, statements, shadowing not allowed.	26
Tiger	Educational functional language, everything is an expression, duplicate declarations allowed, no returns.	17

The sections below describe the tests that we have performed. The code snippets show an original program on the left with the function call highlighted in blue, and the expected transformed program on the right. The corresponding unit tests check if the program returned by our implementation matches the expected program.

In the previous sections we mainly used C++ as the language for code snippets. We continue this trend for the test cases below: test case examples are taken from our C++ tests. The corresponding tests of other languages should only differ in syntax.

Some test cases do not apply to all languages, such as tests relating to the 'return' statement, which does not exist in Tiger. In that case there is no corresponding SPT test for said language. This also accounts for the difference in the number of test cases in Table 5.2.

Note that the tests are not generic, they cannot directly be copied to another language. If a language engineer wants to test inline function for his language, the relevant testing programs have to be translated to that language.

5.2.1 General Cases

The general test cases are to make sure that the implementation is functional in a basic setting. This covers language constructs that can contain a function call, without the added complication of naming conflicts or control flow disruptions.

Among other things, this tests whether the function body is correctly inserted. For example, when the function call is located in the condition of an if-statement, the body statements need to be inserted before the start of the if-statement, see Figure 5.3.

```
1  int foo(int x) {
2    return x + 1;
3  }
4
5  int main() {
6    if (foo(2) < 5) {
7      do_something();
8    }
9  }
```

```
1  int foo(int x) {
2    return x + 1;
3  }
4
5  int main() {
6    int x = 2;
7    if ((x + 1) < 5) {
8      do_something();
9    }
10 }
```

Figure 5.3: The function call is located in the condition of the if-statement, so the inlined body statements should be placed before the if-statement.

The general cases also include expressions where the order of operations changes when the return expression of the function replaces the function call, see Figure 5.4. The full return expression should have the highest precedence when inserted, so it should be wrapped in parentheses when necessary. Figure 5.4 also tests that when multiple calls to the same function are present, only the selected function call is inlined.

```
1  int foo(int x) {
2    return x + 1;
3  }
4
5  int main() {
6    foo(2) * foo(2);
7  }
```

```
1  int foo(int x) {
2    return x + 1;
3  }
4
5  int main() {
6    int x = 2;
7    foo(2) * (x + 1);
8  }
```

Figure 5.4: This test case investigates if the order of operations is preserved. Simultaneously it ensures that the correct function call is being inlined when there are multiple calls to the same function in one expression.

5.2.2 Class Methods

Obviously, class methods are only relevant for object-oriented languages. They can contain function calls, but they can also be the function call that is being inlined. Calls to a class method inside the same class are very similar to regular function calls in the main scope. Figure 5.5 shows this for a method call that does not involve any attributes.

```
1  class MyClass {
2    public:
3      int foo() {
4        return 1;
5      }
6      int bar() {
7        return foo();
8      }
9  }
```

```
1  class MyClass {
2    public:
3      int foo() {
4        return 1;
5      }
6      int bar() {
7        return 1;
8      }
9  }
```

Figure 5.5: A simple example of inlining a method call.

However, the biggest challenge related to classes is correctly dealing with attribute and method references. As mentioned in Section 3.7.3 we are missing a reference generation strategy. Consequently, the implementation is unable to fix a reference to a class attribute/method by prefixing the relevant object. This limits the ability of our implementation to inline method calls located outside the class. In its current state, if a reference to an attribute/method of the class is present in the method body and the method call is outside the class, the inline action is supposed to be rejected — inlining the reference should create an unresolved reference or unfixable capture. Therefore, our tests related to this expect the inline action to fail.

Then there is the case of dynamic dispatch, where a reference has two potential target declarations. However, testing this would mainly test the Statix implementation of the language in question. The scope graph of a program shows us if an identifier can reference multiple declarations. If Statix tells us that there are multiple possible references, the inline function implementation simply has to abort.

5.2.3 Name-Fix

The name-fix algorithm checks for- and fixes variable capture after the AST has been transformed. To test whether the name-fix implementation works as expected we test several programs that cause variable capture when inlining a function call. The most basic form of variable capture is when one of the inserted declarations shadows an existing declaration. Figure 5.6 demonstrates this by inlining `foo(2)` in a lower scope than the existing declaration of `x`. This inserts a shadowing declaration of `x` for the parameter. Name-fix is expected to rename the inserted declaration and its intended references to `x_0`.

```
1  int foo(int x) {
2    return x + 1;
3  }
4
5  int main() {
6    int x = 5;
7    {
8      foo(2);
9      x = 6;
10   }
11 }
```

```
1  int foo(int x) {
2    return x + 1;
3  }
4
5  int main() {
6    int x = 5;
7    {
8      int x_0 = 2;
9      x_0 + 1;
10     x = 6;
11   }
12 }
```

Figure 5.6: Inlining `foo` inserts a declaration of `x` that captures the already existing reference to `x` below the function call. This capture is fixed by name-fix.

If line 9 is not present in the original program (left side of Figure 5.6), the inserted declaration no longer captures a reference and therefore does not have to be renamed. However, in WebDSL it does have to be renamed, since WebDSL does not allow shadowing.

Tiger also differs from C++ here. In Tiger, the parameter declarations and function body are inserted as part of an isolated inner scope (Figure 5.7 shows an inlined function call in Tiger that would cause capture in C++). This means that there is no way for an inserted declaration to capture an existing reference. The only references that can be captured in Tiger are the references from the function body, as described below.

```
1  let
2    var x : int:= 5
3    function foo(x : int) : int =
4      x + 1
5  in
6    foo(2);
7    x := 6
8  end
```

```
1  let
2    var x : int := 5
3    function foo(x : int) : int =
4      x + 1
5  in
6    let
7      var x : int := 2
8    in
9      x + 1
10   end;
11   x := 6
12 end
```

Figure 5.7: An inline function example in Tiger. The resulting program would cause capture in C++, but does not cause capture in Tiger.

It could also happen that an already present declaration captures an inserted reference. For example, a function body can contain references to a global declaration. If that declaration is shadowed in the scope of the function call, inlining the function will cause the reference to be captured by the shadowing declaration. Since the shadowing declaration is capturing, we should rename that and not the inserted reference that is captured. Figure 5.8 shows a program where this happens and what we expect as output in the unit tests. This case of capture also occurs in Tiger. In WebDSL, it only applies to shadowed global declarations because other declarations are not allowed to be shadowed.

```
1  int x = 5;
2
3  int foo() {
4    return x;
5  }
6
7  int main() {
8    int x = 2;
9    foo();
10   x = 3;
11 }
```

```
1  int x = 5;
2
3  int foo() {
4    return x;
5  }
6
7  int main() {
8    int x_0 = 2;
9    x;
10   x_0 = 3;
11 }
```

Figure 5.8: The declaration on line 8 shadows the global declaration of `x`. Inlining `foo()` causes the reference to the global declaration to be captured, but name-fix repairs it by renaming.

Name-fix finds- and renames all declarations that are capturing in the current scope graph. However, this does not necessarily solve all occurrences of capture. Renaming declarations could cause other declarations to capture references. To make sure that those references are repaired too, name-fix recursively calls itself until there is no capture left or an unfixable semantic error occurs. This recursive pattern can be tested with programs where a declaration is shadowed multiple times, see Figure 5.9.

```
int x = 5;

int foo() {
  return x;
}

int main() {
  int x = 1;
  {
    int x = 2;
    {
      int x = 3;
      {
        foo();
      }
    }
  }
}
```

```
int x = 5;

int foo() {
  return x;
}

int main() {
  int x_2 = 1;
  {
    int x_1 = 2;
    {
      int x_0 = 3;
      {
        x;
      }
    }
  }
}
```

Figure 5.9: An inline function case where fixing one occurrence of capture introduces another. Name-fix recursively renames the capturing declarations until none are left.

5.2.4 Static Semantics Errors (Duplicate Definitions)

When static semantics errors occur we either expect a language-specific strategy to fix them or we expect the test to fail. One example that falls under such errors is an unresolved reference that occurs because a declaration is unreachable from the scope of the function call. This should result in failure. Testing a semantic error that must be repaired not only depends on *which* errors the language engineer decides to repair, but also *how* they are repaired. Fixes for a semantic error are implemented per language and not part of our implementation, so we do not test them. However, we implemented a helper strategy that renames when a duplicate declaration error occurs, because duplicate errors are a common static semantics error and closely related to name-fix. Therefore, we do include tests for this helper strategy if the language does not allow duplicate declarations. Figure 5.10 shows one of these tests, where an inserted declaration is renamed to remove the duplication conflict.

```
int foo(int x) {
  return x + 1;
}

int main() {
  int x = 5;
  foo(2);
  x = 6;
}
```

```
int foo(int x) {
  return x + 1;
}

int main() {
  int x = 5;
  int x_0 = 2;
  x_0 + 1;
  x = 6;
}
```

Figure 5.10: The inserted declaration for parameter `x` causes a duplicate definition error in C++. The error is fixed by renaming the declaration and its intended references.

In WebDSL, shadowing is not allowed and raises errors. In our language-specific strategies for WebDSL, we handle this error just like duplicate declarations. Subsequently, inserted declarations that shadow an existing declaration are renamed. This renaming fix is tested by all WebDSL tests where shadowing occurs. This includes some tests related to name-fix and recursive function calls.

Tiger does not raise errors for duplicate declarations: it allows you to re-declare any variable wherever you want. Therefore, there are no duplicate declaration tests for Tiger.

5.2.5 Recursive Function Calls

Recursive function calls are function calls located inside the definition that they reference. Inlining them can cause a special case of capture: an original declaration in the function body can become shadowed by its inserted copy (this does not apply to Tiger because the inserted declarations are part of an isolated inner scope). Such a shadowing inserted declaration might capture a reference to the original declaration. The inline function algorithm fixes this by renaming the inserted declaration and only its intended references. We test this with a program where the described case of capture occurs, as shown in Figure 5.11. The function call on line 3 is inlined creating a declaration that shadows the original parameter declaration. The inserted declaration and its intended references are renamed. Note that in line 3 of the transformed program the occurrence of `x` is not renamed, because it comes from the argument expression in the original program that is meant to reference the original parameter declaration.

```
1   int to_ten(int x) {
2     if (x < 10) {
3       x = to_ten(x + 1);
4     }
5     return x;
6   }
```

```
1   int to_ten(int x) {
2     if (x < 10) {
3       int x_0 = x + 1;
4       if (x_0 < 10) {
5         x_0 = to_ten(x_0 + 1);
6       }
7       x = x_0;
8     }
9     return x;
10  }
```

Figure 5.11: A special case of variable capture can occur with recursive calls. The program on the right shows the renaming fix that is expected.

5.2.6 Void Functions

For void functions the function call expression has to be replaced with an empty expression. What this looks like depends on the language — extracting the return expression and removing the function call is implemented as two language-specific functions — but the way the extracted return expression is passed on throughout the implementation should still be tested. We test this with multiple cases, where the inlined function call is located in different types of enclosing expressions/statements and with empty- and non-empty function bodies, with- and without a return keyword at the end.

In Tiger we consider procedures to be the void functions. Some C++ void function tests do not apply to Tiger, because Tiger procedures are not allowed to be empty: their body must always contain an expression. Additionally, Tiger does not have the return keyword, cancelling tests for the empty return statement.

5.2.7 Unreachable References

Unreachable references are references inside the function body that cannot reference the same declaration from the scope of the function call. This only happens for references to declarations outside the function definition, since the declarations inside the body are inserted together with the reference. The most common case of a reference becoming unreachable is with references to private attributes that are moved to a scope outside the class. Figure 5.12 on the next page shows a method call that cannot be inlined for this exact reason. If a reference becomes unreachable, the inline action should be rejected, so we do not provide an expected output program. Instead, we specify that we expect the inline action to fail.

```
1   class MyClass {
2     private:
3       int x = 2;
4     public:
5       int get_x() {
6         return x;
7       }
8   };
9
10  int main() {
11    MyClass bar;
12    bar.get_x();
13  }
```

fails

Figure 5.12: The private attribute `x` is unreachable from the scope of `main`. Therefore, inlining `get_x()` yields an unreachable reference and should be rejected.

Chapter 6

Discussion and Evaluation

The main difficulty of performing a refactoring is ensuring that the behavior of the program does not change. This is a big motivation for implementing an automatic version of a refactoring: if the implementation itself is behavior preserving, you do not have to manually check it every single time you apply the transformation.

But, it is also important that the refactoring remains accessible. If it requires a lot of effort from the language engineer before it can be used, it is not a big improvement from developing the refactoring on your own for the language in question only.

In Chapter 2 we introduced challenges specifically for inline function and Chapter 3 shows how we tackled them in our algorithm. This chapter reflects on our algorithm and implementation in Spoofax, discussing its correctness, limitations, performance and usability.

6.1 Correctness and Termination

As mentioned in Chapter 1, a refactoring is a transformation to a program that preserves the observable behavior of the program. So in order for a refactoring algorithm to be correct, it must perform the specified transformation on any program where it can be applied and the resulting program must have the same observable behavior as the original. This allows us to discuss the correctness of our inline function algorithm.

In the context of correctness it is important to keep in mind that the algorithm is language parametric. The language parameters are out of our control. Therefore, the correctness of the algorithm is language parametric as well, in the sense that we have to assume the correctness of the language parameters. Because of its language parametric nature, it is difficult to formally prove the correctness. To achieve this, a clear distinction has to be made between the language parameter components whose correctness is assumed and the generic components that require a proof of correctness. In our attempts to perform as many steps in a generic manner as possible (such as dealing with return statements), this distinction has become somewhat obscure. Therefore, the remainder of this section will only reason about the correctness of the algorithm instead of providing a formal proof of correctness as described above.

SPT tests. Chapter 5 explains that we tested the implementation by applying it to function calls in test programs and checking if the output programs match expected refactored programs. The main purpose of these tests is to discover and fix bugs, but they also show that the implementation performs the specified transformation on programs where it can be applied. Now, obviously we cannot guarantee that this covers all possible programs. The implementation is generic, so theoretically there is an infinite number of language constructs that have to be tested. However, by covering the most common constructs (return keywords, shadowing, void functions/procedures, etc.) we argue that we have covered sufficient edge

cases to show that the implementation performs the specified transformation of inlining a function call. Additional test cases can always improve this argument, but because we already cover most (if not all) of the generic components in our tests, it is more likely that the additional test cases will cover language-specific functions rather than generic components of the algorithm.

6.1.1 Behavior preservation

The main challenge of discussing the correctness of our algorithm is showing that it is behavior preserving. In his work on refactorings for object-oriented languages Opdyke defines what it means for a refactoring to be behavior preserving [2]. He introduced certain preconditions that have to hold for a refactoring to be applicable. The precondition he mentioned for function inlining is that in the function body there should be no references to declarations that are unreachable from the scope of the function call. Next to this precondition he mentions that there should be no "name collisions" after the refactoring is applied (Opdyke briefly mentions that name collisions can be fixed by renaming) and return statements need to be converted to branching/goto statements. This is a good starting point for discussing the correctness of our function inlining algorithm.

The Precondition. Our algorithm does not check the precondition beforehand. Instead, it performs the transformation and then analyzes the transformed program using the Statix analysis. This way unreachable references should manifest themselves as unresolved references to the static semantics analysis, which are picked up as errors that cause our algorithm to disallow the inline action for the particular function call. Therefore, although our algorithm does not check the precondition mentioned by Opdyke beforehand, it is still checked before returning the transformed program.

Syntax/Semantic Errors. We already established that we assume the language parameters to be correct. This includes the parser, unparser, static semantics analyzer and language-specific functions. In our algorithm, all insertions and deletions to the program are performed by the language-specific functions to respect the language grammar. With the assumption that the language parameters are correct, this ensures that the transformed programs are at least legal programs. That is, the final transformed AST does not contain syntax- or semantic errors if the language parameters correctly perform their duty.

Name Collisions. When it comes to name collisions, we rely on the correctness of the name-fix algorithm [15] and our check that no inserted declaration is referenced by a non-inserted name. The name-fix algorithm is designed to detect and eliminate capture that takes place in program transformations. The paper on name-fix proves that name-fix eliminates variable capture, and that when applied to a transformed program it yields α-equivalent programs (programs that are equal up to the names of the bound variables).

Name-fix requires two assumptions for this proof to hold. The first assumption is that the static semantics analysis correctly yields a name graph. In our algorithm this translates to the assumption that the static semantics analysis correctly derives the scope graph of the program, which we cover by assuming the correctness of the language parameters.

The second assumption requires the scope graph derived from the program to behave deterministically. This means that for the same program, an identifier that references other identifiers should always map to the same declaration in the scope graph. This is in line with our requirement in Section 3.2: querying the scope graph for a reference should always return the same individual declaration (dynamic binding is not allowed).

With these two assumptions the name-fix paper shows that it eliminates variable capture for a given transformation. In the claim that name-fix correctly eliminates variable capture we refer to the name-fix paper, specifically their correctness section (section 4) and proofs in its appendix.

However, as mentioned in Section 3.7.4, when a function call is present in its own function body, name-fix could miss it when an inserted declaration captures a reference inside the function body. Name-fix copies the ID of the original declaration to the inserted declaration, so references to the original declaration that are captured by the inserted one do not change the name graph. Name-fix does not see this as capture since the copied declaration is considered a valid replacement for the original. However, the data flow of the two declarations can differ. Luckily, this form of capture can only be caused by inserted declarations capturing non-inserted references, because that is the only way for two separate declarations to share one ID. Our algorithm ensures that no inserted declarations are referenced by non-inserted references before applying name-fix, so we can confidently state that our algorithm covers this case of capture as well.

In Section 3.7 we already showed that the remaining cases of name collisions cause errors (such as references not being resolved or duplicate definitions). These static semantics errors are assumed to be detected by the static semantics analyzer and if they are not corrected by a language-specific function, our algorithm aborts the inline action when the errors are detected.

From all this we conclude that our algorithm correctly ensures behavior preservation when it comes to name collisions, with the assumption that the static semantics analysis is correct and behaves deterministically on the input program.

Control Flow (Return Statements). As Opdyke stated, the return statements need to be converted to branching or jumping statements. This is to preserve the control flow of the program. As we mention in Future Work, with a control flow encoding of the language this could be checked generically by comparing Control Flow Graphs (and possibly even repaired, more on this in Section 8.1.3). However, our algorithm does not assume the presence of Control Flow Graphs (yet). Instead, return statements are handled by language-specific functions. The function body is exposed to a language-specific function `ls-fix-control-flow`. This function should, if applicable, at least remove the last return statement. It is up to the language engineer to repair other control flow constructs such as converting early return statements to branching/goto branches.

The language-specific function `ls-is-return` is expected to recognize control flow altering constructs that should have been removed by `ls-fix-control-flow`, such as the 'return' keyword. Our algorithm relies on the function to be able to ensure that all control flow changing constructs have been transformed. For now, this means that the control flow preservation is mainly in the hands of the two aforementioned language-specific functions. However, the algorithm is designed in such a way that the two calls to these language-specific functions can easily be replaced by more generic functions, like functions working on the Control Flow Graph as mentioned in Section 8.1.3.

6.1.2 Termination

When it comes to termination, there are two steps in the algorithm that potentially create a loop. The first one is `ls-fix-semantic-error` that repeatedly analyzes the transformed program and tries to fix the static semantics errors until there are no errors left. Now, we do check that `ls-fix-semantic-error` actually changes the AST before doing a recursive call (if an error has occurred and it does not change the AST we abort). However, after it has changed AST_T to AST'_T to fix an error, a new error may be found whose fix changes AST'_T back to AST_T. This would cause non-termination, but is in the hands of the language engineer, so we can only assume that the language engineer does not introduce this loop.

The second potential loop is in the name-fix step of the algorithm. When name collisions are fixed by renaming name-fix recursively calls itself. Luckily, the name-fix paper contains a proof of termination that we reference here to claim termination.

6.2 Usability of Implementation

The Unit tests in Chapter 5 show that our implementation produces a working Inline Function refactoring for multiple languages. However, we have not yet reflected on its usability. With usability, we refer to the amount of effort required to deploy the implementation. After all, if it takes a lot of effort before a generic refactoring can be used, it might be more appealing for the language engineer to just implement a language-specific version of the refactoring from scratch. In this section we reflect on the usability of our implementation.

The implementation is made in Spoofax. Obviously, this means it can only be used for languages defined in Spoofax. More concretely, this translates to two requirements: a working SDF3 grammar and a static semantics specification in Statix. Writing a grammar and a Statix specification is quite time-consuming and might not be very efficient if all you want to do is use the inline function refactoring. However, a Spoofax implementation brings a lot of extra features to your language, such as Eclipse editor plugins, access to Stratego to create new transformations and access to other refactorings like renaming. With additional features like these a Spoofax implementation can certainly be worth considering. Besides, to define a behavior preserving function inlining implementation you will need a parser and static semantics analyzer one way or another, so if you do not have one already, writing a Spoofax implementation might not be a bad idea after all.

With the Renaming refactoring of Spoofax, an SDF3 grammar and a Statix analysis is all that is required. Once you have those, you can use the refactoring simply by adding a single API call to a Stratego library. This is where Inline Function deviates from Renaming, with its need for a set of language-specific functions.

Chapter 3 introduced a set of language-specific strategies and Section 4.4 describes them in more detail. Requiring such strategies means that the language engineer has to put in some extra work before the refactoring can be used. Now, this could partially be avoided. With a fixed standard for AST structures (labels, subterm order, etc.), some language-specific strategies could be generalized for all Spoofax languages that adhere to that standard. For example, by fixing a function definition term to have the structure and labels as displayed in Figure 6.1, the language-specific strategy `ls-exctract-data` that extracts the subterms of a function definition can be generalized.

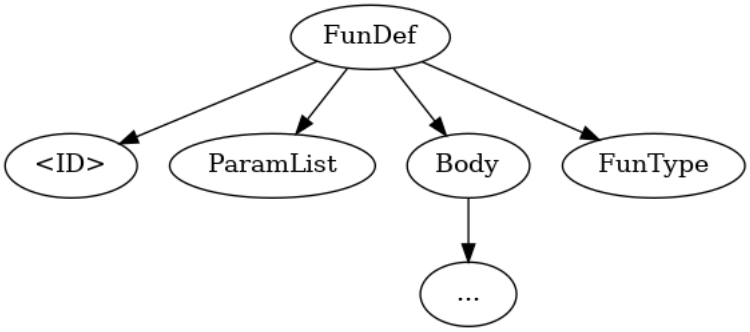


Figure 6.1: Example of a fixed AST structure for function definitions.

However, fixing the AST structure greatly reduces the freedom of designing your own language, whereas a major argument for building a new (Domain Specific) Language in Spoofax is the freedom to design a custom grammar to suit your specific needs. So, as long as you want to maintain this freedom, there is no way around the requirement of language-specific strategies.

Luckily, most language-specific strategies are straightforward to implement. For example, extracting subterms is a simple pattern match on their parent term. The language-specific code we had to write for C++ is less than 100 lines in size, roughly half of which consists of

comments and whitespacing. For Tiger it even takes less than 50 lines, with each language-specific function consisting of only a single line.

Furthermore, strategies like inserting a block of statements may be relevant for many other refactorings, such as unrolling a loop or moving statements into a function [1]. This indicates that language-specific strategies could be re-usable for other refactorings. Eventually when enough language parametric refactorings are added to Spoofax's core, less and less new language-specific strategies are required because they have already been defined for other refactorings.

Comparing this to defining Inline Function from scratch for your Spoofax language, we see that it takes less effort to use our generic implementation. The language-specific functions are needed in both situations. For example, how are you going to inline a function call from scratch without extracting the function body from the function definition? This already means that creating your own Inline Function implementation from scratch will take at least as long as deploying our implementation. And on top of that, with the generic implementation you do not have to think about variable capture, re-analyzing the transformed AST, performing all steps in the right order, etc. With a clear understanding of the language in question and the purpose of each language-specific function, deploying Inline Function for an existing Spoofax language should take roughly one hour to complete. Therefore, we can conclude that the generic implementation is usable in practice.

6.2.1 Ease of access

It is also important that the refactoring is accessible once it has been deployed. Spoofax generates an Eclipse editor plugin that invokes our implementation. After opening a program file in Eclipse, a user can select a function call by highlighting it. Then it can be inlined by pressing the 'Inline Function' button in the Spoofax Refactoring toolbar. The user receives details from the inline action in the form of readable GUI popup messages (Section 4.5.1), including debugging information if an error occurred (the relevant input of a failed strategy). The error messages not only improve user experience but can also aid the language engineer in debugging the language-specific functions.

The single button press and informative error printing make for an accessible tool. On top of that it supports usage for programs that are distributed over multiple files, automatically renames identifiers to eliminate capture and in the future it will preserve comments and whitespacing with Spoofax's layout preserving unparser. All in all, we argue that our implementation is an accessible automatic refactoring.

6.3 Performance

IDEs are used to improve the process of software development. They provide all kinds of useful features, ranging from syntax highlighting to code completion. These features have to analyze the program and run some instructions in order to produce their output. If there is a significant time delay in this process, then the workflow of the programmer could be disrupted. Therefore, the code of these features should be as efficient as possible. For a refactoring this is no different: nobody wants to wait a full minute for a function to be inlined. Therefore, it is important that we analyze the performance of our implementation in terms of response time.

6.3.1 Complexity

When discussing the runtime it makes sense to start with the theoretical worst-case complexity. In the context of complexity the lines of code of the input program can be used to express the input size. Larger programs have larger ASTs that take longer to traverse and analyze,

which are the most significant operations of our refactoring. For example, the generation of NameFixIDs requires a loop through all ASTs of the project, visiting every identifier. This takes much more time than operations like matching for a return statement in the function body or combining parameters and arguments into variable declarations. In total there are 3 loops through the ASTs *before* name-fix is called that, in a worst-case scenario, all have to traverse the full AST.

However, there is a much more significant step in our implementation: the fact that name-fix contains recursive calls. Every time name-fix changes the AST to fix an issue, it recursively calls itself, restarting the AST traversals and Statix analysis. Potentially every single identifier in the program captures an identifier after inlining the function call, causing many recursive calls. Consequently, the worst-case complexity of the name-fix process is more significant than any other preceding steps of the implementation.

Luckily, it is highly unlikely that such a worst-case scenario occurs in practice. If capture is detected, all occurrences of capture in the current state of the program are fixed in a single recursive call. The only way that there are multiple recursive calls is if new occurrences of capture appear by fixing the previous occurrences of capture. This requires many nested scopes with variable declarations that all have the same name.

Following this reasoning, the performance of the implementation depends on three things: Firstly, the size of the input program, which affects the runtime of the traversals and analysis. Secondly, the number of static semantics errors and name collisions that occur by inlining the function call. This affects the number of recursive calls in name-fix. And thirdly, the efficiency of the Statix analysis.

The Statix analysis is a language parameter, so we do not control its efficiency. However, it is expected to be the most complex step of our algorithm. After all, to derive a scope graph from a program, the analysis needs to traverse the full program, deriving reference relations and checking all other static semantics constraints. Depending on the size of the program and the present language constructs, this yields a much greater complexity than the other steps of our implementation.

6.3.2 Runtime in Practice

The complexity section above suggests that the runtime bottleneck of the implementation should mostly be the name-fix part (including the Statix analysis of the transformed AST). Both mainly depend on the total number of identifiers of the program rather than the size of the function definition that is being inlined, since they have to explore the reference relations of the identifiers.

To test our hypotheses we run the C++ and Tiger implementation on dummy code files with a set number of lines of code (LOC). The first half of the files consists of single-line function definitions, the other half consists of one call per function definition. This way, there are many reference relations that have to be explored by the Statix analysis and by our name-fix approach. In our choice for the dummy files we took inspiration from a compiler benchmarking tool[1]. We generate the dummy files using a python script that can be found in our github repository[2]. The tests were executed using Spoofax 2.5.16 on a Windows 10 Education system with an AMD Ryzen 7 5800X processor.

In our initial experiments we also tested the runtime when inlining larger function definitions without adding identifiers (for example, in the 1000 LOC file we added a sequence of 500 constant expression statements to the function definition that is being inlined) but we observed a minimal increase in runtime (less than 1% for the 1000 LOC example with a 500 LOC function definition), so we deemed the single-line function definition structure sufficient for the performance tests.

[1] https://github.com/nordlow/compiler-benchmark

[2] https://github.com/MetaBorgCube/function-inlining

To measure the runtime we inline the last function call in each dummy file and store the runtime of the full refactoring, the name-fix step and the analysis step inside name-fix. The initial experiments showed that the coefficient of variation (the result of dividing the standard deviation by the average) is less than 1% after only 5 runs, so we decided that the average of 5 runs is a reliable result for every file. The results can be seen in Table 6.2 and Table 6.3, where 100 LOC (lines of code) means 50 function definitions and 50 function calls.

Table 6.2: This table shows the performance results when inlining the last function call in the C++ dummy files. In the dummy files, half of the LOC consist of function definitions, the other half of function calls to each definition in a main function. All function definitions, including the one being inlined, consist of a single expression.

C++				
LOC	**Total Runtime**	**Name-fix %**	**Analysis %**	**Coefficient of Variation**
100	0.87s	93.55%	84.10%	0.010
200	1.90s	94.10%	79.03%	0.045
300	3.39s	95.17%	77.31%	0.008
400	5.66s	95.94%	77.37%	0.005
500	9.48s	96.81%	78.43%	0.009
600	13.61s	97.37%	78.89%	0.003
700	19.51s	97.89%	80.11%	0.009
800	27.18s	98.15%	81.86%	0.005
900	36.11s	98.50%	82.44%	0.020
1000	47.51s	98.64%	83.47%	0.005

Table 6.3: The performance results for Tiger, similar to the results in Table 6.2.

Tiger				
LOC	**Total Runtime**	**Name-fix %**	**Analysis %**	**Coefficient of Variation**
100	0.33s	99.99%	59.76%	0.026
200	0.91s	98.91%	46.61%	0.010
300	1.75s	98.86%	40.48%	0.024
400	2.86s	98.95%	36.08%	0.008
500	4.19s	99.09%	33.46%	0.007
600	5.86s	99.25%	31.00%	0.017
700	7.71s	99.35%	29.71%	0.015
800	9.86s	99.37%	28.84%	0.009
900	12.35s	99.37%	27.61%	0.012
1000	15.01s	99.45%	26.87%	0.014

The first observation is that the runtime clearly scales with the program size. For example, when going from 500 LOC to 1000 the runtime more than quadruples for C++ and triples for Tiger. More specifically, the runtime seems to scale with the number of identifiers, since every additional line of code in the dummy files adds identifiers.

We also see that for both C++ and Tiger, the name-fix process takes up almost all the runtime (up to 99%). This is what we expected, because it includes the Statix analysis of the transformed program, loops through the name-graphs several times and has to query the scope graph for every identifier in the program. This also means that if we want to improve the runtime of the implementation we should look at the name-fix part first.

Another observation is that the C++ experiments take a lot more time than the Tiger experiments. As shown by the Analysis column, this difference can largely be attributed to the Statix analysis of C++; if we remove the runtime of the analysis for both C++ and Tiger their total runtimes are a lot closer. This indicates that the Statix analysis of Tiger is simpler

and more efficient.

The fact that the Statix Analysis takes such a long time for C++ demonstrates that the runtime is language parametric: roughly 80% of the runtime comes from the Statix analysis, a language parameter. However, this still leaves around 20% for the rest of the name-fix process, which is around 8 seconds for C++ and 11 seconds for Tiger. The 3 seconds difference here can be explained by the function calls in the experiments. Tiger's function calls have three references:

```
sum := sum + function-call()
```

Whereas the C++ experiments only have two references:

```
sum += function-call()
```

Depending on how frequently you need to apply the refactoring, 8 to 11 seconds could be an acceptable response time. However, when extrapolating the runtimes for program sizes larger than 1000 LOC the runtime quickly exceeds an acceptable response time for a refactoring. In short, this means that for large projects the current state of the implementation is unsuitable, unless you are very patient.

6.3.3 Where to look for improvements

Since over 98% of the runtime is caused by the name-fix operations, improving the performance of the implementation needs to start by improving the name-fix process. The performance of the Statix analysis is out of our hands, so we need to look at the remaining operations. After inspecting the runtime of the individual operations we discovered that `check-capture` takes up most of the time (in Tiger's 1000 LOC experiment, over 10 seconds of the 11 seconds in name-fix are spent in `check-capture`). `check-capture` is the strategy that checks for capture according to the definition of capture from the name-fix paper [15]. Optimizations there would improve the performance drastically. In fact, it might even make sense to have a built-in optimized name-fix library in Spoofax. It is relevant to many refactorings and maybe even other transformations, because it checks for behavior preservation in terms of reference relations.

Chapter 7

Related work

The research area of language parametric refactorings is relatively small and most research on refactorings only implements transformations for a select number of languages. Nonetheless, there is some interesting research that is worth mentioning. This section lists novel and influential research that is related to the topic of this thesis.

Renaming For Everyone [21]. Misteli's master's thesis is in many ways the predecessor to our thesis. His thesis presents a language parametric algorithm for the rename refactoring, also implemented in Spoofax. The Renaming refactoring mainly involves finding all related occurrences that have to be renamed, and checking that no reference relations have been changed using name-fix. The key difference between rename and inline function is that renaming does not require knowledge of the structure of the AST: in an AST, all identifiers are assumed to be string terms without a language-specific constructor name. No terms have to be deleted or inserted, which means you do not have to bother with retaining a valid AST structure according to the grammar. The only language-specific strategies needed for renaming are the parser and the static semantics analysis.

Refactoring Object-Oriented Frameworks [2]. Opdyke's PhD thesis was on a set of automated refactorings specifically designed for object-oriented languages. A large part of his thesis is dedicated to the fact that refactorings need to be behavior preserving. He lists seven properties that have to hold for a refactoring to be behavior preserving. The first six properties are not relevant for inline function, but the seventh property regards "Semantically Equivalent References and Operations", which is closely related to- but more general than the checks in the name-fix part of our algorithm.

In his thesis Opdyke lists a multitude of what he calls "low-level refactorings". He formalizes preconditions for these refactorings and reasons about their process of execution. One of the refactorings he addresses is "inline function call". The only precondition he defines for it is that all declarations referenced from inside the function body need to be reachable from the scope of the function call. In his description of the process of function inlining after checking the preconditions, capture is addressed by the name of "name collisions". He states that some variable names from the function body may need to be renamed to avoid these collisions, but does not go into detail. Finally, he mentions that return statements need to be converted to branch/goto statements, but again does not go into more detail.

Practical Analysis for Refactoring [26]. Roberts' PhD thesis extends Opdyke's research [2] in multiple ways. Where Opdyke mainly specified formal preconditions for his "low-level" refactorings, Roberts goes one step further and defines postconditions. Furthermore, Roberts introduces primitive functions that can analyze specific components of programs and together can be used to ensure valid refactorings. In particular the function called "Method", which returns a method node belonging to a "selector", shows similarities to the language-specific strategy `ls-extract-data` from our set of language-specific strategies. The

thesis does not include the inline function refactoring.

Towards Generic Refactoring [27]. The first mention we could find of language parametric refactorings is by Lämmel. In his paper, Lämmel specifies a refactoring framework that is, as he calls it, ”largely language-independent”. The paper lists commonalities between languages, for example the general notion of scopes. It also recognizes that there may be a need for language-specific ingredients for the definition of refactorings. The framework specifies generic functions and function combinators that aid in the development of a generic refactoring, similarly to what Stratego and Statix do for us. Using those functions the framework provides several generic algorithms for simple analysis and transformations. Additionally, the framework defines an interface for abstractions of a language, much like we use Stratego strategy arguments for language-specific strategies.

Static Composition of Refactorings [28]. Kniesel and Koch developed a formal model for combining existing transformations to create larger transformations. Although their main focus is on refactorings, the larger concept of transformations is also included, arguing that many refactorings consist of some non behavior preserving transformation steps. The paper does not formally define generic refactorings, only a model that can combine existing refactorings to create new ones. The model is evaluated and tested with a practical implementation for Java. The idea to separate a refactoring into smaller transformation steps is relevant to our approach of inline function. For example, we split removing the return statements, inserting the code block and applying name-fix into three separate steps.

Inline Function Expansion for Compiling C Programs [29]. Long before refactorings started being used to improve readability they were employed to optimize compiled code. This paper by Hwu and Chang investigates cases in which function inlining can improve performance of a compiled program. It also addresses issues with inline function and cases in which it could change the behavior of the program. The paper is the earliest papers we could find on function inlining.

Specification, implementation and verification of refactorings [30]. Schäfer worked out concepts and techniques for a modular specification of refactorings. His work focuses on the object-oriented language Java. He investigates the relevance of name binding and control- and data flow for refactorings. Schäfer implements the three refactorings ”Rename”, ”Extract method” and ”Inline Temp” for Java, following the techniques presented in the paper. The ”Inline Temp” refactoring inlines a variable declaration.

Refactoring Haskell Programs [31]. Whereas most of the research mentioned above concerns object-oriented languages, Li investigates refactorings for the functional programming language Haskell. He found that a select number of refactorings including renaming behave similarly when applied to function languages compared to object-oriented languages, but for the other refactorings there are significant differences, for example in their analysis of program components. He also argues that due to the ”clean semantics of functional languages and a rich theoretical foundation for reasoning about programs”, it is convenient to show that a refactoring is behavior preserving for Haskell.

Li describes the process of ”Unfold a definition”, a refactoring that is essentially the same as inline function and inline variable combined. However, he does not go into detail about the specification and verification of unfolding a definition. Additionally, Li briefly addresses Lämmel’s work on language independent refactorings and concludes that it is impossible to be completely language independent. This is in line with the language-specific strategies that are required for our algorithm.

Chapter 8

Conclusion

This thesis presents a language parametric algorithm for the inline function refactoring. It describes an implementation of the algorithm in the language workbench Spoofax and tests the implementation in multiple languages using Unit tests to show that it works.

The algorithm depends on three language parameters: an AST parser, a static semantics specification and a set of language-specific functions. Using these parameters, the algorithm defines readily available refactoring steps and an interface with a single API call that combines them into the full refactoring. It finds the referenced function definition in the AST and extracts the parameters and arguments to create variable declarations. Return statements are dealt with and a code block is constructed that is inserted into the AST. Finally, the algorithm checks for static semantics errors and name capture, and attempts to repair them by several means including the name-fix algorithm by Erdweg et al. [15]

The algorithm is implemented in Spoofax. Because of its language parametric nature, it does not have to be rewritten to be deployed for a different language. The implementation has been tested for three languages with various static semantics, showing that it can be used in practice. The implementation is available for use in any Spoofax language. Deploying it requires the definition of a set of language-specific strategies, which should take roughly one hour to do, assuming that you know the (static) semantics of the language in question.

8.1 Future Work

The current state of the algorithm still has limitations. For example, it rejects function definitions that contain multiple return statements. Possible solutions to limitations and other future work is addressed in the subsections below.

8.1.1 Layout Preservation

Section 4.1 mentions that the layout-preserving unparser [18] implemented in Spoofax is not working due to an issue with the origin tracking [19] implementation. We tried to determine the source of the issue and found that the implementation of the Stratego primitive `get-parent` is not working when invoked on an analyzed AST. Misteli's implementation of renaming can successfully use the layout-preserving unparser, because if the only thing that changed in the AST is a string, then the part of the unparser that calls `get-parent` is never reached. Once the issues of the unparser are fixed it would be useful to use it in the implementation of inline function.

8.1.2 Reference Generation

Section 3.7.3 already introduced a possible approach to extending a referencing term such that it can reach the same declaration from a different scope. An example of when this is

relevant is when a class method is called on an object of the class, say `myObj.foo()`, and the method call is being inlined. If `foo()` references an attribute, say `myAttr`, the reference only works inside the class. So when it is moved to the scope of the method call it needs to be prefixed with the object to preserve the reference (`myObj.myAttr`).

Now, we have a scope graph at our disposal that encodes the connections between scopes with edges. The edges can be explored to determine a path from a scope to a declaration. By mapping the types of edges to AST terms, we could generate a correct reference to the intended declaration. This will require a language-dependent strategy that converts scope graph paths to terms, however the hardest part is the exploration of the tree, which can be implemented generically.

8.1.3 Control Flow Validation and Generation

The control flow semantics of a language concern the order in which instructions of a program are executed. Spoofax can express the control flow using FlowSpec [32], a declarative specification language for dataflow analysis. With a FlowSpec specification, you can analyze the control flow of programs. This returns a Control Flow Graph (CFG). The original intention of FlowSpec is to use the CFG for dataflow analysis, however we could use it to validate that the control flow has not changed by checking for something like bisimulation equivalence [33] between the CFG of the original program and the transformed program.

Furthermore, it might be possible to design a control flow generation algorithm similarly to the reference generation algorithm from Section 8.1.2. The CFG shows the control flow in the form of edges between certain points in the program. If we know the conditions that have to apply for an edge to be traversed, we could translate the control flow graph of a function body to branches/jumps. This way, the control flow of early-return statements as mentioned in Section 3.5 could be mimicked semi-generically.

Our algorithm is designed in such a way that it can easily be adapted to use the two CFG functions described above. The call to `ls-fix-control-flow` can be replaced by the control flow generation algorithm and the call to `ls-is-return` can be replaced by a function that compares the CFG of the original and transformed AST.

8.1.4 A Language-Specific Strategy Suite

Roberts [26] already mentioned the need for some primitive functions that can analyze a program for the validation of multiple different refactorings. Similar to his primitive functions, our language-specific strategies can be useful for other refactorings than inline function (Section 6.2). For some, it might be possible to define a more general strategy that is applicable to a larger set of refactorings. Obviously, the strategies would still need to be implemented by the language developer, but receiving multiple refactorings in return instead of just one is a much better incentive for putting in the work to define them.

8.1.5 Combining Refactorings

Kniesel et al. [28] defined a formal model that specifies if- and how smaller refactorings can be combined to create larger refactorings. Their model is intended to work program parametrically, so it should work for all programs of a specific language. It might be possible to extend this model to a language parametric model, where it could be applied to refactorings defined in Spoofax.

Some programmers interpret inlining a function as inlining *all* occurrences of the function call and then removing the function definition. In case our inline function refactoring would be combined with a refactoring that removes unused definitions to create this larger inlining refactoring, recursive functions should be approached with special care. Our current implementation of inline function will insert the function body that contains a recursive

call. This could create a non-termination situation where it keeps inlining and inserting the recursive call.

Bibliography

[1] Martin Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley object technology series. Addison-Wesley, 1999. ISBN: 978-0-201-48567-7. URL: http://martinfowler.com/books/refactoring.html.

[2] William F. Opdyke. "Refactoring Object-Oriented Frameworks". PhD thesis. Urbana-Champaign, IL, USA: University of Illinois, 1992.

[3] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. "Automated Behavioral Testing of Refactoring Engines". In: *TSE* 39.2 (2013), pp. 147–162. DOI: 10.1109/TSE.2012.19. URL: http://doi.ieeecomputersociety.org/10.1109/TSE.2012.19.

[4] Brett Daniel et al. "Automated testing of refactoring engines". In: *ESEC/FSE*. 2007, pp. 185–194. DOI: 10.1145/1287624.1287651. URL: http://doi.acm.org/10.1145/1287624.1287651.

[5] Melina Mongiovi. "Scaling testing of refactoring engines". In: *OOPSLA*. 2016, pp. 15–17. DOI: 10.1145/2984043.2984048. URL: http://doi.acm.org/10.1145/2984043.2984048.

[6] Lennart C. L. Kats and Eelco Visser. "The Spoofax language workbench: rules for declarative specification of languages and IDEs". In: *OOPSLA*. 2010, pp. 444–463. DOI: 10.1145/1869459.1869497. URL: https://doi.org/10.1145/1869459.1869497.

[7] Markus Völter et al. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. ISBN: 978-1-4812-1858-0. URL: http://www.dslbook.org.

[8] Arie van Deursen, Paul Klint, and Joost Visser. "Domain-Specific Languages: An Annotated Bibliography". In: *SIGPLAN* 35.6 (2000), pp. 26–36. DOI: 10.1145/352029.352035. URL: http://doi.acm.org/10.1145/352029.352035.

[9] Luis Eduardo de Souza Amorim and Eelco Visser. "Multi-purpose Syntax Definition with SDF3". In: *SEFM*. 2020, pp. 1–23. DOI: 10.1007/978-3-030-58768-0_1. URL: https://doi.org/10.1007/978-3-030-58768-0_1.

[10] A. V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, Aug. 2006.

[11] Hendrik van Antwerpen et al. "Scopes as types". In: *PACMPL* 2.OOPSLA (2018). DOI: 10.1145/3276484. URL: https://doi.org/10.1145/3276484.

[12] Arjen Rouvoet et al. "Knowing When to Ask: Sound scheduling of name resolution in type checkers derived from declarative specifications (Extended Version)". In: Zenodo, Oct. 2020. DOI: 10.5281/zenodo.4091445. URL: http://doi.org/10.5281/zenodo.4091445.

[13] Martin Bravenboer et al. "Stratego/XT 0.17. A language and toolset for program transformation". In: *SCP* 72.1-2 (2008), pp. 52–70. DOI: 10.1016/j.scico.2007.11.003. URL: http://dx.doi.org/10.1016/j.scico.2007.11.003.

[14] Lennart C. L. Kats, Rob Vermaas, and Eelco Visser. "Integrated language definition testing: enabling test-driven language development". In: *OOPSLA*. 2011, pp. 139–154. DOI: 10.1145/2048066.2048080. URL: http://doi.acm.org/10.1145/2048066.2048080.

[15] Sebastian Erdweg, Tijs van der Storm, and Yi Dai. "Capture-Avoiding and Hygienic Program Transformations". In: *ECOOP*. 2014, pp. 489–514. DOI: 10.1007/978-3-662-44202-9_20. URL: http://dx.doi.org/10.1007/978-3-662-44202-9_20.

[16] JW Lloyd. "Practical advantages of declarative programming". English. In: *Unknown*. Conference Proceedings/Title of Journal: Joint Conference on Declarative Programming. 1994, pp. 3–17.

[17] Paul Klint. "A Meta-Environment for Generating Programming Environments". In: *TOSEM* 2.2 (1993), pp. 176–201. DOI: 10.1145/151257.151260. URL: http://doi.acm.org/10.1145/151257.151260.

[18] Maartje de Jonge and Eelco Visser. "An Algorithm for Layout Preservation in Refactoring Transformations". In: *SLE*. 2011, pp. 40–59. DOI: 10.1007/978-3-642-28830-2_3. URL: http://dx.doi.org/10.1007/978-3-642-28830-2_3.

[19] Arie van Deursen, Paul Klint, and Frank Tip. "Origin Tracking". In: *JSC* 15.5/6 (1993), pp. 523–545.

[20] Stephen A. Edwards. *Tiger Language Reference Manual*. Columbia University. 2002.

[21] Phil Misteli. "Renaming for Everyone: Language-Parametric Renaming in Spoofax". MA thesis. TU Delft, May 2021. URL: https://repository.tudelft.nl/islandora/object/uuid:60f5710d-445d-4583-957c-79d6afa45be5?collection=education.

[22] Marjan Mernik, Jan Heering, and Anthony M. Sloane. "When and how to develop domain-specific languages". In: *ACM Comput. Surv.* 37.4 (2005), pp. 316–344. DOI: 10.1145/1118890.1118892. URL: http://doi.acm.org/10.1145/1118890.1118892.

[23] Tobi Vollebregt, Lennart C. L. Kats, and Eelco Visser. "Declarative specification of template-based textual editors". In: *LDTA*. 2012, pp. 1–7. DOI: 10.1145/2427048.2427056. URL: http://doi.acm.org/10.1145/2427048.2427056.

[24] Pierre Néron et al. "A Theory of Name Resolution". In: *ESOP*. 2015, pp. 205–231. DOI: 10.1007/978-3-662-46669-8_9. URL: http://dx.doi.org/10.1007/978-3-662-46669-8_9.

[25] Hendrik van Antwerpen et al. "A constraint language for static semantic analysis based on scope graphs". In: *PEPM*. 2016, pp. 49–60. DOI: 10.1145/2847538.2847543. URL: http://doi.acm.org/10.1145/2847538.2847543.

[26] Donald B Roberts. *Practical Analysis for Refactoring*. Tech. rep. USA, 1999.

[27] Ralf Lämmel. "Towards Generic Refactoring". In: *corr* cs.PL/0203001 (2002). URL: http://arxiv.org/abs/cs.PL/0203001.

[28] Günter Kniesel and Helge Koch. "Static composition of refactorings". In: *SCP* 52 (2004), pp. 9–51. DOI: 10.1016/j.scico.2004.03.002. URL: http://dx.doi.org/10.1016/j.scico.2004.03.002.

[29] Wen-mei W. Hwu and Pohua P. Chang. "Inline Function Expansion for Compiling C Programs". In: *PLDI*. 1989, pp. 246–257.

[30] Max Schaefer. "Specification, implementation and verification of refactorings". British Library, EThOS. PhD thesis. University of Oxford, UK, 2010. URL: http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.580861.

[31] Huiqing Li. "Refactoring Haskell programs". British Library, EThOS. PhD thesis. University of Kent, UK, 2006. URL: http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.555284.

[32] Jeff Smits and Eelco Visser. "FlowSpec: declarative dataflow analysis specification". In: *SLE*. 2017, pp. 221–231. DOI: 10.1145/3136014.3136029. URL: http://doi.acm.org/10.1145/3136014.3136029.

[33] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. ISBN: 978-0-262-02649-9.

Acronyms

AST Abstract Syntax Tree

DSL Domain-Specific Language

IDE Integrated Development Environment

CFG Control Flow Graph

SDF3 Syntax Definition Formalism 3

SPT SPoofax Testing language

Appendix A

Language-Specific Strategies — C++

```
module inline-function-cpp20

imports
  analysis
  pp
  injections/c-syntax-injections
  inline-function-source
  inline-function-name-fix

rules
  // Separate rule for SPT tests.
  inline-function-test(|selected-term): ast -> <inline-function(ls-extract-data, ls-make-decs,
    ls-fix-control-flow, ls-is-return, ls-construct-block, ls-is-exp-language, ls-is-statement,
    ls-insert-before, ls-remove-exp, ls-fix-semantic-errors, editor-analyze, multi
    |<new-iset>)> (selected-term, ast)

  // At the time of writing, construct-textual-change doesn't work because of issues with
  // origin tracking. In case it is fixed, replace pp-debug with construct-textual-change.
  inline-function-menu-action = inline-function-action(pp-debug, ls-extract-data, ls-make-decs,
    ls-fix-control-flow, ls-is-return, ls-construct-block, ls-is-exp-language, ls-is-statement,
    ls-insert-before, ls-remove-exp, ls-fix-semantic-errors, editor-analyze, multi
    | "inlined.cpp")

  // Change to "id" to enable- and "fail" to disable multi-file mode.
//  multi = id
  multi = fail

  // Match on function call terms, returning the full term, name and arguments.
  ls-extract-data(|()): t -> (t, name, args)
    where
      <?Call(Var(NamespaceID(_, name)), args)> t
  ls-extract-data(|()): t -> (t, name, args)
    where
      <?Call(Proj(_, name), args)> t
  // Match on function definition terms, returning the parameters, body and type.
  ls-extract-data(|name): FunDef(type, Declarator(_, FunDeclTyped(IdDecl(name),
      ParamList(params))), _, body)
    -> (params, body, type)
```

```
39
40  // Turn parameter definitions and arguments into variable definitions.
41  ls-make-decs: (params, args) -> decs
42    where
43      decs := <zip(\ (ParamDecl1(type, name), value)
44                     -> Decl(type, [InitDecl(name, value)]) \)>
45                (params, args)
46
47  // Get rid of the last return statement. Also extract the return expression.
48  ls-fix-control-flow(|[Void()]): Compound(body) -> (body', ())
49    where
50      (body', Return(NoExp())) := <split-init-last> body
51  ls-fix-control-flow(|[Void()]): Compound(body) -> (body, ())
52  ls-fix-control-flow(|_): Compound(body) -> (body', return-exp)
53    where
54      (body', Return(return-exp)) := <split-init-last> body
55
56  // Match on return statements.
57  ls-is-return = ?Return(_)
58
59  // Combine variable definitions and a list of statements into a new list of statements.
60  ls-construct-block: (body, params) -> Compound(<concat> [params, body])
61
62  // Indicate that this is a statement language.
63  ls-is-exp-language = fail
64
65  // Match on statement terms.
66  ls-is-statement = is-cpp20-BlockItem-or-inj
67
68  // Insert a list of statements before the argument statement in the AST.
69  ls-insert-before(|stmt, block): ast -> ast-t
70    where
71      ast-t := <oncetd(insert-before-helper(|stmt, block))> ast
72  // We reject inlining functions in the condition of a loop construct.
73  insert-before-helper(|For(_, _, _, _), _): _ -> ()
74  insert-before-helper(|ForEach(_, _, _, _, _), _): _ -> ()
75  insert-before-helper(|While(_, _), _): _ -> ()
76  insert-before-helper(|DoWhile(_, _), _): _ -> ()
77  // In case the function body is empty, nothing needs to be added in front of the statement.
78  insert-before-helper(|stmt, Compound([])): stmt -> stmt
79  // There are cases where the statement is a singular statement, so it is not part of a list.
80  // In that case it should be turned into a Compound block when the other statements are added.
81  insert-before-helper(|stmt, Compound(new-block)): stmt -> Compound(<conc> (new-block, [stmt]))
82  // Else, the new statements should just be inserted in the list containing the specified stmt.
83  insert-before-helper(|stmt, Compound(new-block)): t -> new-t
84    where
85        (before, after) := <split-fetch(?stmt)> t
86      ; new-t := <conc> (before, new-block, [stmt], after)
87
88  // In C++ the function call only needs to be removed for void functions.
89  // Void functions can't be part of an expression, so they will always be standalone expression
90  // statements.
```

```
91  // Therefore we just remove the statement containing the function call.
92  ls-remove-exp(|exp): ast -> ast-t
93    where
94      // split-fetch ensures that the element is in the list.
95      // If we would use filter instead, the strategy would succeed for all lists.
96      ast-t := <oncetd(<conc> <split-fetch(?Exp(exp))>)> ast
97
98  // other-asts equals [] unless multi-file mode is enabled and there are other programs in
99  // the project. It is passed on in case the other files are needed to fix a semantic error.
100 // If it is filled, it contains (ast, tuple) entries.
101 ls-fix-semantic-errors(|analysis-errors, (old-ast, old-analysis), asts-analyses):
102   new-ast -> fixed-ast
103   where
104       (duplicate-term, _) := <fetch-elem(where(Snd; string-ends-with(|"already defined")))>
105         analysis-errors
106     ; fixed-ast := <rename-duplicate-declaration> ((old-ast, old-analysis),
107                                                    asts-analyses, new-ast, duplicate-term)
```

Appendix B

Language-Specific Strategies — Tiger

```
module inline-function-tiger

imports
  pp
  analysis
  injections/Tiger-injections
  inline-function-source

rules
  inline-function-test(|selected-term): ast -> <inline-function(ls-extract-data, ls-make-decs,
    ls-fix-control-flow, ls-is-return, ls-construct-block, ls-is-exp-language, ls-is-statement,
    ls-insert-before, ls-remove-exp, ls-fix-semantic-errors, editor-analyze, multi
    |<new-iset>)> (selected-term, ast)

  // At the time of writing, construct-textual-change doesn't work because of issues with
  // origin tracking. In case it is fixed, replace pp-debug with construct-textual-change.
  inline-function-menu-action = inline-function-action(pp-debug, ls-extract-data, ls-make-decs,
    ls-fix-control-flow, ls-is-return, ls-construct-block, ls-is-exp-language, ls-is-statement,
    ls-insert-before, ls-remove-exp, ls-fix-semantic-errors, editor-analyze, multi
    | "inlined.tig")


  // Change to "id" to enable- and "fail" to disable multi-file mode.
//  multi = id
  multi = fail

  // Match on function call terms, returning the full term, name and arguments.
  ls-extract-data(|()): t -> (t, name, args)
    where
      <?Call(name, args)> t
  // Match on function definition terms, returning the parameters, body and type.
  ls-extract-data(|name): FunDec(name, params, type, body) -> (params, body, type)
  ls-extract-data(|name): ProcDec(name, params, body) -> (params, body, ())

  // Turn parameter definitions and arguments into variable definitions.
  ls-make-decs: (params, args) -> decs
    where
      decs := <zip(\ (FArg(name, type), value) -> VarDec(name, type, value) \)> (params, args)
```

```
39
40    // There is no return statement in Tiger, so we don't do anything here.
41    ls-fix-control-flow(|_): body -> (body, ())
42
43    // There is no return statement in Tiger so a return match should always fail.
44    ls-is-return = fail
45
46    // Combine the body expression and variable definitions into a "Let" term.
47    ls-construct-block: (body, []) -> body
48    ls-construct-block: (body, params) -> Let(params, [body])
49
50    // Indicate that this is an expression language.
51    ls-is-exp-language = id
52
53    // The three language-specific strategies below are only required for a statement language.
54    ls-is-statement = fail
55    ls-insert-before = fail
56    ls-remove-exp = fail
57
58    // We don't know of any semantic errors in Tiger that we can fix here.
59    ls-fix-semantic-errors = fail
```