A Mechanized Theory of Aspects

Von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur
Erlangung des akademischen Grades
eines Doktors der Ingenieurwissenschaften (Dr.-Ing.)
genehmigte

D i s s e r t a t i o n

vorgelegt von

Diplom Informatiker
Henry Sudhof

aus Frankfurt am Main

Promotionsausschuss:

Vorsitzender:	Prof. Dr.-Ing. Uwe Nestmann
Berichter:	Prof. Dr.-Ing. Stefan Jähnichen
Berichter:	Prof. Michael Mendler, PhD
Berichter:	Florian Kammüller, PhD

Tag der wissenschaftlichen Aussprache: 28.05.2010

Berlin 2010

D83

Abstract

Over the past ten years, Aspect Orientation has emerged as a new paradigm in software engineering. The key feature promoted in this paradigm is the ability to modularize concerns – aspects – that by nature or implementation are orthogonal to the modular structure of the base application. To allow such a modularization, these aspect modules themselves are able to cut across the structure of the underlying application. This cutting across the structure goes beyond the mere referencing of remote data, allowing changes of the base application's control flow. Thus, Aspects offer the functionality of calling themselves instead of the intended target of an invocation. This altered invocation of the aspect is able to alter, extend or even replace parts of the original program.

Being able to drastically alter the base application's structure comes at the price of a decrease in safety: aspects themselves do not fulfill the expectations held for classical modules and can even harm the modularization of the existing application. Basic concepts such as the locality of errors, type soundness and re-usability can thus be harmed by aspects.

This thesis applies the method of rigorous language development on Aspect Orientation in order to show what classes of aspects maintain the safety of an application. Our rigorous approach entails the design of a family of aspect-oriented core calculi, entirely mechanized in the interactive theorem prover Isabelle/HOL. The emphasis of the work is placed on two different fields. Fundamental questions of modularity, type soundness and subtyping form the first such field. Technical considerations such as binders, variable representation and code generation are the content of the other. In the field of language metatheory, the thesis contributes a comparative me-

chanization of the ς_{Asc} calculus, comparing the locally nameless and the de Bruijn variable representations.

Features of the ς_{Asc} calculi developed in this thesis are a modular concept for aspects in an object oriented setting and several type systems building upon that foundation. These type systems enforce static type safety on aspects, while maintaining modularity and thus re-usability. This approach is also used to handle the variance issues encountered when combining aspects with depth subtyping. Furthermore, several classes of aspects are identified, including a class of compositional aspects.

Zusammenfassung

In den vergangenen zehn Jahren hat sich Aspektorientierung als ein neues Paradigma in der Softwaretechnik etabliert. Dieses Paradigma erlaubt die Modularisierung von Funktionalitäten, die orthogonal zur eigentlichen Modulstruktur eines Programms liegen. Um dies zu ermöglichen, sind solche Aspektmodule selbst nicht an die Modulstruktur gebunden, sondern in der Lage diese zu durchbrechen. Dieses Durchbrechen geht über Referenzieren bestehender Strukturen hinaus, vielmehr verändern Aspekte den Kontrollfluss der ursprünglichen Anwendung und verursachen so ihre eigene Ausführung. Dieser invertierte Kontrollflusses bietet somit die Möglichkeit, Teile des ursprünglichen Programms zu ergänzen, zu entfernen oder zu ersetzen.

Diese Fähigkeit geht mit einem Sicherheitsverlust einher: Aspekte erfüllen oft nicht die Erwartungen an Module und können die Modularisierung der eigentlichen Basisanwendung sogar stören. Grundsätzliche Anforderungen an Module wie Fehlerlokalität, Typsicherheit oder Wiederverwendbarkeit sind somit für Aspekte beziehungsweise mit ihnen nicht unbedingt erfüllt.

Die vorliegende Arbeit behandelt die Thematik rigoros, indem eine Familie von aspektorientierten Kalkülen vollständig in dem interaktiven Theorembeweiser Isabelle/HOL entwickelt wird. Dazu werden zunächst die Grundlagen mechanisierter Sprachtheorie eingeführt und bestehende Ansätze zur formalen Behandlung von Aspekten verglichen. Der darauf aufbauende wissenschaftliche Beitrag gliedert sich in zwei Bereiche: Den der formalen Behandlung der Aspektorientierung und den der mechanisierten Sprachtheorie. Im ersten Feld besteht der Beitrag dieser Arbeit zunächst in der Entwicklung eines aspektorientierten Kernkalküls auf Grundlage des ς Kalküls. Wichtige Kerneigenschaften des Kalküls werden bewiesen und es wird ein Kompositionalitätsbegriff für Aspekte entwickelt.

Auf dem Gebiet der mechanisierten Sprachentwicklung stellt diese Arbeit ein objekt- und aspektorientiertes Kalkül vor, das vollständig in dem interaktiven Theorembeweiser Isabelle/HOL realisiert wurde. Auf Basis dieses Kalküls wurden zudem zwei verschiedene Variablenrepräsentationen erprobt und gegenübergestellt. Dieser direkte Vergleich bietet einen weiteren Beitrag für die Mechanisierung von Sprachen.

Hauptgegenstand der Arbeit ist die statische Analyse von Aspekten mittels Typsystemen, zunächst mittels eines einfachen modularen Typsystems, dann in iterativen Schritten um Subtyping und Varianz erweitert. Jedes vorgestellte Typsystem zur Typisierung von Aspekten umfasst einen kompletten Beweis der Typsicherheit. Ergebnisse des Kalküls werden im Wege einer Codegenerierung und einer Szenarienanalyse auf die Ebene der Anwendbarkeit übertragen. Darauf aufbauend, erfolgt die Entwicklung einer Kategorisierung von Aspekten aufgrund ihrer Sicherheitseigenschaften.

Contents

CHAPTER 1

Introduction

This thesis presents a mechanized formalization of an aspect-oriented calculus that formally establishes the safety properties of the new paradigm Aspect Orientation. New programming languages and concepts have been among the most frequently used tools to master the software crisis since its initial discovery [Naur and Randell, 1968]. The development of original language concepts usually begins with a phase of added flexibility, when a novel idea offers a method to solve issues such as developing large programs, maintaining old software or accelerating development. Often, such concepts are developed without establishing a formal basis at their beginnings. The resulting implementations lack the safety properties expected from mature languages and require a long development process until nuances are well understood and formally established. Common examples of such nuances are typing issues, modularity and concurrent behavior.

When establishing properties of systems, especially formal systems such as programming languages, the mathematical proof is the single most accepted tool. However, proofs themselves do not guarantee completeness, or even correctness. Furthermore, presenting a proof in a notation understandable for readers requires the omission of numerous steps. Conversely, the inclusion of said steps introduces a requirement to validate the proofs that is to check every single step, making proofs hard to read and follow. Theorem Provers, or to be more specific, "proof assistants" or "interactive theorem provers" [Wiedijk, 2006], Isabelle, Coq, HOL or PVS being popular examples, were designed for this very purpose [Kammüller, 2006]. By mechanically checking each step in a proof, they establish a formal method of

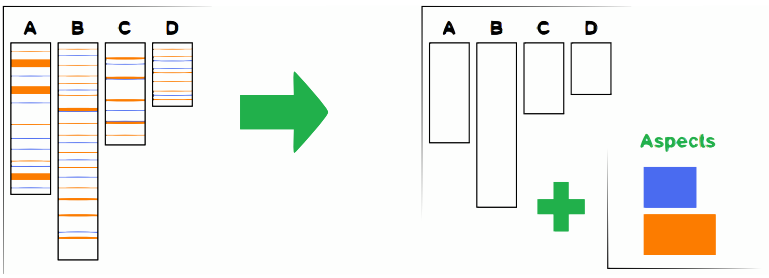

Figure 1.1: Separation of concerns: Aspects modularize cross-cutting concerns.

executing and presenting complete proofs. The rigorous development and verification of programming language concepts is one of the foremost applications of interactive theorem provers. The strictness and versatility of such a mechanized formalization guarantees that new concepts introduced into programming languages are valid and have no unanticipated effects. The more expressive power a change to a language has, the more it calls for a rigorous approach for validation.

Aspect Orientation (AOP) [Kiczales et al., 1997] has recently emerged as a particularly powerful concept that is especially popular as an extension in object-oriented programming. Its promise to cross-cut the established structure of programs also implies the ability to cross-cut the properties of the host language. This puts fundamental concepts like type safety and basic security at risk. Figure 1.1 gives a high-level schematic of the idea behind AOP.

In this dissertation, we present an aspect-oriented calculus based on the Abadi and Cardelli Object calculus [Abadi and Cardelli, 1998]. Our calculus was entirely formalized in the Interactive Theorem Prover Isabelle/HOL and establishes a type safe approach to aspects. Through this approach, we were able to establish a theory of compositional aspects.

Our reason for performing research on the foundations of Aspect Orientation lies in the seemingly paradox nature of its paradigm. The paradox can be demonstrated as follows: Aspect Orientation heavily emphasizes flexibility. The control required to constrain flexibility, however, does not derive directly from the concept. This situation introduces problems into the practice of Aspect Oriented Programming that prevent the widespread use of the technique as they contradict established expectations of software engineering.

Aspects have the ability to drastically change program semantics and thus have an effect on the modular structure of programs. Thus far, ques-

tions regarding the safety and soundness of aspect-oriented programs have not been sufficiently answered by the existing formal approaches for object-oriented programs. Our goal is to determine precisely how flexible AOP can (safely) be and what constraints are required to maintain safety.

The need for such work is visibly demonstrated by the sample code in Figure 1.2. That code is not type safe. Yet, this fact is not caught by AspectJ's [Kiczales et al., 2001; AspectJ, 2009] typechecker.

```
public class Test
{
  public Test test()
  {
    return this;
  }
}
 public aspect asp
{
  Object around() : call(* *.test(..))
  {
    return "oops";
  }
}
```

Figure 1.2: A not type safe program in AspectJ.

In the example, we use a feature introduced by Aspect Orientation: the ability to overwrite methods based on certain criteria. In this case, the aspect will introduce a method body that returns a `string` instance instead of the expected `Test` – a clear type error. The program will terminate with a `ClassCastException` in `Test.test`. This breaches modular reasoning, as the base class Test is sound on its own.

A problematic characteristic of aspects is that the quintessential quantification over oblivious programs is not well-suited for modular reasoning. AOP encourages the concept of having aspects only loosely linked to the points of their application and joinpoints not linked to their aspects at all. Such a pattern, however, deviates from the established patterns of structured programs.

In our opinion, AOP requires a concept to enable modular reasoning on a strict formal basis to improve re-usability and modularity [Sudhof, 2006]. This will necessarily entail a careful and rigorous re-examination of its definitions. Object Orientation grew popular because of well-typed approaches

on both, the practical and theoretical level. AOP is still one step short of reaching the level of Object Orientation.

We argue that AOP's boundaries must be established if it is to become a solid concept for manageable flexibility. These boundaries, furthermore, must be grounded in a rigorously proven formal basis. Such a formal basis would be valuable when devising new and advanced formal approaches, for instance in the field of static security analysis. AOP is often criticized [Steimann, 2006; Constantinides et al., 2004] for its lack of stability. The work described in this thesis strives to a) precisely show the cause of some problems of AOP, and b) to show that a safe subset of AOP exists.

We see the solution to establishing such a formal basis in designing, formalizing, mechanizing and verifying a core calculus realizing aspect-oriented features. The idea behind a core calculus is to use a bare-bones mathematical model, the λ calculus being a well-established example, to establish properties of much larger, real-world languages [Barendregt, 1984]. This means that the core calculus must share certain fundamental properties with the languages of the domain in order to allow the analysis of the domain essential features in a strict setting, that is a relation must be established to provide a translation between the concepts. This translation can stem from obviously shared concepts, case studies, example translations or even full compilations (i.e. morphisms) from one concept to the other.

1.1 Intent of this Thesis

The main intent behind this thesis is to establish a sound concept of Aspect Orientation that has been proven in a rigorous environment. This core goal unifies two major lines of research. The first line is to research the safety and soundness of Aspect Orientation, its reality, its fundamental limitations and the viable properties. Our goal here is to maintain modularity, especially static type safety, while also allowing a loose coupling of aspects and base programs. The second line of research in this thesis is the rigorous realization of the above concept in an interactive theorem prover. Interactive theorem provers are an important tool for the development of languages and their application in this domain is of key interest for the field of language metatheory [Aydemir et al., 2007]. To the best of our knowledge, there is no mechanized aspect-oriented core calculus in existence[1].

[1] We present a short survey of existing non-mechanized AOP calculi in Section 3.

1.2 Structure and Reading Hints

This thesis is structurally divided into two parts that can be read independently of each other. The first part constitutes an introduction into the foundations for the second part and is intended to establish a common base for all readers, regardless of their various backgrounds. The second part presents the results of our research, the aspect-oriented core calculus ς_{Asc} in its various incarnations.

For reference, Figure 1.3 offers a visual representation of the dependencies between the chapters.

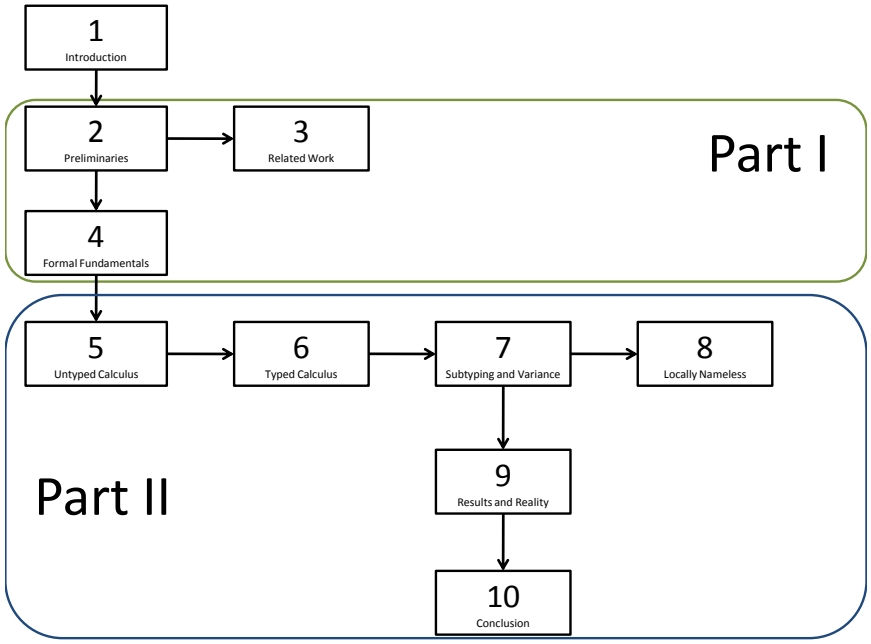

Figure 1.3: Structure of this thesis.

Part I begins with a short introduction into the domain of this thesis in Chapter 2. The chapter briefly summarizes basic vocabulary of Aspect Orientation in Section 2.1. It continues with a short survey of core calculi suitable for reasoning about object-oriented languages, namely the λ [Church, 1940; Barendregt, 1984] and ς [Abadi and Cardelli, 1998] calculi and Featherweight Java [Igarashi et al., 1999] in Section 2.2. As a conclusion to this background chapter, interactive theorem provers, especially Isabelle and Coq, are presented in Section 2.3.

In Chapter 3 we present a survey of core calculi for Aspect Orientation. After presenting the approaches, we offer an extended discussion about the differences, strengths and weaknesses of the various calculi in Section 3.4.

The final chapter of Part I, Chapter 4 gives a quick overview of the various steps required to formalize and mechanize a language or calculus. This overview begins with straightforward tasks such as formalizing syntax and semantics in Section 4.1, before briefly handling the topic of types and type soundness in Section 4.3. The chapter then continues with a discussion of the disciplines of the POPLmark Challenge [Aydemir et al., 2007]. The first part of this challenge, the highly specialized problem of representing binders and variables, is covered in Section 4.4, with an emphasis on the locally nameless and de Bruijn index approaches. Two small sections, Section 4.5 and Section 4.6, about induction and code generation respectively, then conclude the chapter.

Part II begins with the presentation of our untyped calculus ς_{Asc} in Chapter 5. The chapter opens in Section 5.1 with the basic formalization of our calculus and the technicalities of its mechanization. The addition of these aspects to the calculus is covered in Section 5.2 with the definition of aspects and weaving. Proven results of this basic formalization are collected in Section 5.3, especially the compositionality of aspect weaving and the confluence of the calculus. The chapter closes with an extended discussion of the approach in Section 5.4. As this chapter introduces the details of the mechanization shared by the following extensions of this basic approach, it is of considerable importance.

Types are added to ς_{Asc} in Chapter 6. Following the brief Section 6.1, we continue with the realization of a type system in Section 6.2. We then extend the type system by providing a modular, polymorphic typing notion for aspects in Section 6.3. As we did in the preceding chapter, we offer a collection of proven theorems and properties in Section 6.4. After the introduction of simple types, the chapter finishes with an extension of the type system to accommodate subtyping, forming $\varsigma_{Asc<:}$, in Section 6.5.

Variance issues are the main topic of Chapter 7, with a focus on the covariance issues present in established aspect-oriented languages. The chapter begins with a short introduction into variance, and which cases can be safely allowed, in Section 7.1. Section 7.2 then covers the construction of an extension to the calculus to allow such variance issues to be analyzed: $\varsigma_{Asc<:+}$. Using the extended calculus, we demonstrate a type system able to cope with variance in a safe and sound manner in Section 7.3, explaining why a naïve implementation of subtypes would fail. We close this chapter by formally re-establishing the properties of the calculus in Section 7.4.

Chapter 8 marks a pronounced detour from the narrative of the thesis. Instead of presenting a new extension of the $\varsigma_{Asc<:+}$ calculus, it introduces a re-mechanization of $\varsigma_{Asc<:+}$. The difference is that the version presented in this chapter uses the locally nameless approach to represent variables, instead of the de Bruijn indices used in the mainline chapters. This is a contribution

to the field of language meta-theory, showcasing a modern approach. The chapter presents the adaption of the syntax and semantics in Section 8.1, followed by the type system in Section 8.2.

The critical connection between our formal calculus and reality is established in Chapter 9. This chapter combines several concepts for relating the calculus developed here to real-world situations. The first concept established is code generation: In Section 9.1, we present the results of using the code generation function of Isabelle/HOL. Small case studies are the second approach for connecting to real languages: We show a number of situations in real languages and equivalent formulations in ς_{Asc} in Section 9.2. The results of these case studies are then used to present a classification of aspects in Section 9.3. Finally, we relate our approach of using labels and label interfaces to practically oriented approaches in Section 9.4, showing commonalities and possible ways of connecting them to our research.

The final chapter offers a conclusion to this thesis by collecting the results. Insights into the lessons learned during the research for this thesis and venues for future research are presented, along with much deserved thanks.

Part I

Preliminaries for Language Theory in Interactive Theorem Provers

CHAPTER 2

Background

This part introduces the background of the research presented in this thesis. Starting with Aspect Orientation and Core Calculi in this chapter, it will continue to present related work in Chapter 3. After that, the theory and practice of formalizing languages using interactive theorem provers will be introduced in the third and final chapter of this part.

This first chapter of Part I provides a condensed background of the basic concepts used in this thesis. The first concept is Aspect Orientation, which is covered in Section 2.1. Then, core calculi as the chosen level of abstraction are presented in Section 2.2. Finally, interactive theorem provers as the tools to be used are introduced in Section 2.3.

2.1 Aspect Orientation

Aspect Orientation (AOP) [Kiczales et al., 1997] is a relatively new paradigm in programming. It is seen as an solution to the growing problems encountered when maintaining and developing new systems, as it allows both the adaption of existing code as well as the use of new modularity concepts. With this combination, it is widely viewed as a possible answer to the modern incarnation of the software crisis, which formed the very foundation of Software Engineering as a discipline [Naur and Randell, 1968]. Aspect Orientation facilitates flexibility and modularization in programs by adding the ability to modularize concerns *cross-cutting* the established structure of a program. This has also been adapted as the core of a direction in software

engineering – Aspect Oriented Software Development – advocating the use of new modularity concepts to improve the building blocks of software design [Filman et al., 2005; Sokenou et al., 2006].

The most frequently used definition of AOP [Filman and Friedman, 2005] mentions two essential properties: Being able to *quantify* over places in a program and to adapt these places with code (*advice* in the AOP vocabulary) while keeping the main program *oblivious* to this process. Together, the two properties quantification and obliviousness are seen as the definition of AOP. This definition is not uncontested, but nonetheless widely accepted as a litmus test. Its implications can be better understood by taking this line about the mechanics of adapting into account:

> In program P, whenever condition C arises, perform action A.
> [Filman and Friedman, 2005]

Terms used throughout this thesis are taken from the dominant language implementing the paradigm: AspectJ [AspectJ, 2009]. AOP, as implemented in AspectJ, involves dividing the program into the "plain" Java *base* and implementing *crosscutting-concerns* as *aspects*. The quantification part of the definition takes place in *pointcuts*, which are predicates capturing certain conditions in a program. Action A is the Aspect's code, usually called *advice*, which alters the oblivious *base* program P.

Aspects are usually realized as constructs containing pairs of *pointcuts* and *advice*. *Pointcuts* are expressions describing sets of *joinpoints*, where joinpoints denote exposed sections in the base code. Practically all aspect-oriented languages use methods as the level of granularity for exposed joinpoints. The most established style for denoting pointcuts is to use lexicographic patterns – strings with wildcards – that match the names of the methods used as joinpoints and which can be combined using logical operators.

```
call(void Point.move*(..))
call(void Point.move*(..)) && cflow(call(void Rectangle.move*(..)))
```

Figure 2.1: An AspectJ pointcut selecting all methods named `move`. The '`*`' is a wildcard character, matching any classname.

These primitive lexicographic patterns can be used with a number of advanced operators. The example in Figure 2.1 describes a pointcut that selects all invocations of methods that have names starting with `move` on instances of the class `Point`. The second line adds a condition that only those invocations happening in the context of an invocation to a method starting with `move` of a class named `Rectangle` should be selected.

A common distinction for pointcuts is *static* versus *dynamic*. Static pointcuts refer to a constant set of joinpoints where the question of whether or not to adapt can be answered statically. Dynamic pointcuts, on the other hand, rely on dynamic conditions, such as the content of the stack. Pointcuts by themselves do not alter anything; this is where the advice comes in, the code that details what should happen when a pointcut matches. Like pointcuts, advice comes in several flavors. Specifically, the order of the advice execution can be chosen relative to the base joinpoint. This means that advice can be set to act before the joinpoint's instructions via the `before` modifier. Conversely, `after` performs the advice after the advised joinpoint. The notable case is `around`, which replaces the original joinpoint. In the latter case, the keyword `proceed` is used to return to the original control flow.

Another differentiation between pointcuts are semantics. Predominantly the two kinds, `call` and `execution` are used. The former captures the call of a joinpoint – a method – on the caller's side, while the latter captures the callee's side.

The example below presents a simple aspect with a pointcut and advice:

```
around (call(Object Point.move*(..))) {
         return proceed();
       }
```

The example advice above replaces all calls to methods captured by its pointcut, i.e. all calls to methods of the class `Point` with `move` at the beginning of their name. The `proceed` statement forwards to the original call, realizing an identity function.

It can be easily seen that around advice is capable of simulating before and after advice: Before advice can be placed above the `proceed`, after advice beneath it. From a safety standpoint, the around advice is the interesting case, as it can drastically alter the programming language's properties. For instance: By returning a value of a type not conforming to the original's declaration, type safety can be breached, as shown in Figure 2.1 earlier in this chapter.

The process of combining base and aspect code is universally referred to as *weaving*. Combining the code during compile time is called compile-time weaving. Conversely, the more fine-grained and more modular approach of adapting the code during run-time is called runtime-weaving, with load-time weaving being a possible step in between. The equivalence – or lack thereof – of different weaving scenarios is one of the issues investigated in this thesis.

For reference, we include this small glossary of AOP terms:

advice The executable code in an aspect.

aspect A module consisting of a pointcut and advice.

base The original program.

joinpoint A point in the base program where aspects can be woven.

obliviousness The notion that the base application does not know its aspects.

pointcut The part of an aspect that specifies where the advice should be introduced.

quantification The ability of an aspect to express where its advice should act.

weaving The process of introducing the advice of an aspect into a base application, according to the aspect's pointcut.

2.2 Core Calculi

The concept behind core calculi is rooted in the idea that a very small and simple calculus can be used to perform formal proofs about a much larger and complex system. The key qualification is to capture a sufficient subset of the big system in the calculus, so that a plausible relation to the original, complete concept is maintained. For Object Orientation and derived concepts, three families of calculi are frequently used today, each having the maturity to be used in a rigorous setting.

λ Calculus

The λ calculus [Church, 1936, 1940; Rosser, 1982; Barendregt, 1984] is arguably the most established formal calculus in computer science. It forms the foundation of many modern concepts, including type systems, programming languages, interactive theorem provers and much more.

There are three different reduction conversions defined for the classical λ calculus, which also exist in similar ways for other functional calculi. These are α, β and η conversion. Each conversion expresses a notion of equality between the terms. The α conversion denotes the renaming of variables and will be introduced in Section 4.4. The η conversion encodes extensionality, i.e. the equivalence of $\lambda.fx$ and f, provided that x is not a free variable in f. In the context of semantics, the β conversion – also β reduction – is the most

interesting, as it encodes equality regarding evaluation and thus encodes the reduction relation for terms.

The λ calculus features unparalleled simplicity to encode complete expressive power, as it is solely comprised of functions and their application. This massive expressivity of the pure calculus led to the development of several typed variants. Their purpose was to limit the calculus to manageable subsets, usually with strong properties like normalization. One such variant, System F [Reynolds, 1974; Girard, 1972] – the λ calculus with polymorphic types – , forms the subject of the POPLmark challenge [Aydemir et al., 2007].

While the expressive power of the λ calculus is sufficient to express almost any concept, the complexity of such encodings is non-trivial. To be specific, the λ calculus uses functions as its only construct, which makes the representation of objects awkward. For this reason we do not consider the λ calculus viable for this thesis.

Featherweight Java

Featherweight Java [Igarashi et al., 1999; Pierce, 2002] is a object-oriented core calculus. It is remarkable because it is a functional subset of the programming language Java. This means that every term in Featherweight Java is also a valid Java program, causing the authors of Featherweight Java to coin the line "Inside every large language is a small language struggling to get out." to describe its construction.

Like Java, Featherweight Java uses a nominal type system. Nominal type systems discriminate types by name, not by their properties. Thus, two types can be identical in every detail but for their name and would still be different. By contrast, structural type systems equate types with identical feature sets. However, the subset used to form the calculus can be confusing for users of the "big" Java language, as the functional style required to express programs is very unlike the practice for Java programs. For instance, objects in this formalism do not have a mutable state, but can only have the values that were assigned to them in their constructor. Due to this design decision, there are no reference semantics, but solely value semantics. Summarizing, we consider Featherweight Java to suffer from a pronounced proximity to the language Java.

ς Calculus

The ς family of calculi [Abadi and Cardelli, 1998] is a formal model for the analysis of Object Orientation. A very simple model of objects with the

ability to bind a variable to "self", the surrounding object, forms the family's core concept. The initial plain calculus is iteratively expanded in a series of calculi to introduce concepts including typing and imperative programming [Abadi and Cardelli, 1995] among others.

The basic ς calculus was built on the idea that objects can be described as sets of labeled methods, i.e. named methods. Thus a method has a name and a body, where the body can consist of any ς term. The special object-oriented nature of the calculus is rooted in the way methods use variables: A method has only one "self" variable, which is then replaced by the value of the surrounding object upon invocation. Beyond method invocation, the calculus has only one additional operation: Update.

In the plain ς calculus, there is no discrimination between methods and fields and there are no variables other than self – parameters are passed to methods by updating other methods/fields of the surrounding object before invoking the method. Then, the method itself can access these updated values via the – replaced – self variable. Figure 2.2 gives a short summary of the ς calculus' syntax along with a semi-formal description of its semantics.

Let $o \equiv [l_i = \varsigma(x_i)b_i^{i\in 1..n}]$ (l_i distinct)
o is an object with method names l_i and methods $\varsigma(x_i)b_i$
$o.l_j \rightarrow b_j\{x_j \leftarrow o\}(j \in 1..n)$ selection / invocation
$o.l_j \Leftarrow \varsigma(y)b \rightarrow [l_j = \varsigma(y)b, l_i = \varsigma(x_i)b_i^{i\in(1..n)-j}](j \in 1..n)$ update / override

Figure 2.2: The primitive Semantics of the ς calculus as introduced in [Abadi and Cardelli, 1994].

The ς calculus is highly extensible and was shown to be able to accommodate concepts like reference semantics, class based languages and advanced typing concepts [Abadi and Cardelli, 1998]. While the class-based Featherweight Java might be considered a more natural fit to the typing issues found in AspectJ-related languages, it should be noted that the ς calculi's structural types are able to reflect typing issues in a form analogous to the ones found in real-world applications with a smaller overhead.

The combination of being object-oriented, small and extensible makes the ς calculus the ideal choice for the formalization of Aspect Orientation. Its simplicity makes a mechanized model viable, while the extensibility means that new concepts - aspects - can be introduced in a natural fashion. As the scope of this work is static compositionality and type safety, concepts such as exceptions, concurrency and reference semantics are not required, making the ς calculus a near-perfect fit. Other concepts that share some of these properties, most notably Featherweight Java, are designed to follow particular programming languages too closely, which can render results less

general. On the other hand, the ς- calculus is better suited than the λ calculus, which would require cumbersome embeddings of objects.

2.3 Interactive Theorem Provers

The term "interactive theorem prover" or "proof assistant" describes a family of automated proof tools. Unlike SAT[1] solvers or other fully automated provers, the main intention of interactive provers is not the automatic verification of theorems, but rather the verification of manual proofs. In order to do this, interactive provers employ a process, in which a proof is developed by using commands altering the proof state through the application of *tactics*, much like using inference rules in a manual proof – hence the designation "interactive". Important examples for interactive theorem provers are Isabelle, Coq, PVS and the HOL family (Hol4, HOL light). Most proof assistants strive towards a style that approaches that of classical proofs on paper, while also exploiting the benefits of having a mechanized version of the formalization – namely re-use and code generation. Historically, proof assistants evolved from LCF [Gordon, 2000; Paulson, 1990], the logic of computable functions. This approach implemented a theorem proving environment based on an unpublished logic by Dana Scott[2]. In fact, the following description provided by Milner [Milner, 1972] still holds mostly true for modern systems:

> The proof-checking program is designed to allow the user interactively to generate formal proofs about computable functions and functionals over a variety of domains, including those of interest to the computer scientist – for example, integers, lists and computer programs and their semantics. The user's task is alleviated by two features: A subgoaling facility and a powerful simplifier.

Although the usability of modern systems is far superior to the modest beginnings of Stanford LCF, the fundamental concept of interactive theorem proving has not changed by a substantial degree.

Today, the most important interactive theorem provers follow in the LCF tradition, but use different logics as their foundation. The most popular examples are either based on Russell and Whitehead's simply typed set theory [Whitehead and Russell, 1910] (Isabelle/HOL, HOL) or the Curry-Howard correspondence (Coq).

[1] Solvers for the boolean satisfiability problem, a np complete problem.
[2] The logic was eventually published in 1993 [Scott, 1993].

Another important design fundamental of theorem provers, realized in Isabelle and Coq, is the kernel approach. Provers such as Isabelle or Coq, use their respective basic logics to realize a subset of their logic in a manually verified kernel, usually in a few thousand lines of code. The actual prover implementing the complete logic is then implemented around that kernel, which is used to verify the correctness of its own extensions. In fact, all proofs are rooted in the logic defined by underlying kernels, while users can remain completely oblivious to their basics and limitations.

The most important benefit of using an interactive theorem prover is the strong guarantee of correctness it provides. To be able to do this, provers check and thus guarantee the correctness of all proofs performed on the initial model. In some cases, this added verification of a proof can be the only means to guarantee the correctness of complicated proof. A prominent example for such a proof exceeding the human ability to follow reasoning is the flyspeck project [Mackenzie, 2005; Flyspeck, 2009], aimed at verifying the proof of Kepler's conjecture.

This section introduces Isabelle/HOL in detail while also providing a short introduction to Coq. HOL is omitted, as the logic and features are very similar to Isabelle/HOL, but specialized towards hardware verification. PVS is not considered here, as it does not use the trusted kernel approach and is thus not directly comparable to Isabelle or Coq. For a more complete survey of interactive theorem provers, we refer to Freek Wiedijk's book *The 17 Provers of the World* [Wiedijk, 2006].

Isabelle/HOL

Isabelle is more of a framework for interactive theorem provers than just one isolated theorem prover. In fact, it was originally introduced as being "*the next 700 theorem provers*" [Paulson, 1988]. It is jointly developed by the University of Cambridge and the Technische Universität München, with its roots located at the University of Cambridge and – via LCF – Stanford. Isabelle itself is based on a small subset of intuitionistic Higher Order Logic [Paulson, 1989] – just the meta-conjunction and meta-implication – , but supports a wide array of "object logics", that allow users to perform proofs in different logics. The most popular examples are the Russell/Whitehead Higher Order Logic [Whitehead and Russell, 1910, 1912, 1913] in Isabelle/HOL and – albeit to a lesser degree – the Zermelo-Fraenkel set logic in Isabelle/ZF and the combination of HOL and the logic of computable functions in Isabelle/HOLCF. Isabelle in its current form is fairly user-friendly, aiding mechanizations both by enabling the use of notations very similar to the style one would use on paper and through a powerful interface – the Proof General [Kleymann and Aspinall, 2009].

For this work the Higher Order Logic (HOL) instantiation of Isabelle (Isabelle/HOL) was used. Higher Order Logic is often described as the combination of functional programming and logic [Nipkow et al., 2002]. Advantages of using Isabelle/HOL in particular include the ability to extract code from the formalization to yield executable results from the very model that was verified by the theorem prover. The formalization itself is aided by the extensive library of theorems provided by the Isabelle distribution.

Developing a formalization in an interactive theorem prover like Isabelle/-HOL introduces an overhead when compared to performing the equivalent formalization on paper. This is due to the rigorous approach required to establish a proof in a checked formal model. To reduce this overhead, Isabelle provides strong automatic proof methods and support for using SAT solvers and model checkers. Nonetheless, these methods cannot and should not remove the additional overhead – mechanizing takes several times longer than a plain pen and paper formalization.

Even when setting aside the already mentioned advantages of correctness and code extraction, it should be noted that a certain modularity can be achieved in mechanized proofs, allowing the re-use of portions of models in different settings. This eases development when migrating a formalization of a language to a more powerful type system.

As noted before, Isabelle's kernel is purely based on a small fragment of intuitionistic logic. Isabelle/HOL realizes Higher-Order Logic on that foundation and adds many features on all levels. Examples for such features are essential parts of the formalization presented in this thesis, such as datatypes, inductive definitions and primitive recursive functions [Berghofer and Wenzel, 1999].

Datatypes are a very straightforward construct and establish simple set-theoretic recursive product types. Datatype definitions consist of constructors which themselves consist of *admissible* parts. A part is admissible if it follows a number of rules [Berghofer and Wenzel, 1999], which limit the recursive occurrences. It is admissible to use the datatype itself recursively and mutually recursively with another datatype, provided that the elements stay injective. Moreover, a datatype can be defined as polymorphic using type variables. The example below shows a simple datatype representing the natural numbers.

```
datatype NaturalNumber =
   Zero
  |Suc NaturalNumber
```

Isabelle/HOL automatically creates lemmas for structural induction and case distinction for datatypes. These tie in with a concept built directly on

top of datatypes: Primitive recursive functions, short `primrec`. The concept of primitive recursion realized in Isabelle allows a subset of the primitive recursive functions and is implemented on top of the datatype theory. The use of primitive recursive functions can best be shown by the following example, which realizes the addition on the natural numbers defined above:

```
primrec
  add::NaturalNumber × NaturalNumber ⟹ NaturalNumber
where
   "add Zero x = x"
  |"add (Suc x) y = add x (Suc y)"
```

The function definition has one equation for each constructor of the variable used in the pattern – in this case the first variable. Only that variable is split into its possible constructors; moreover, it is clear that the variable becomes smaller with each iteration, guaranteeing termination. Primitive recursive functions, like all functions in HOL, have to be total.

The concept of inductive sets or predicates is a more basic concept and thus allows more freedom. Inductive sets are defined by a set of inductive rules. To be precise, the inductive set specified by an inductive definition is the least closed set[3] that satisfies the definition's rules. Note that membership in a set is logically equivalent to satisfying a predicate, so that inductive definitions can be read both as definitions of sets and as logical predicates. See the example below for an inductive definition of the set of even numbers.

```
inductive even::NaturalNumber ⟹ Bool
where
  Even_zero "even Zero"
  |even_step "⟦even x⟧ ⟹ even Suc (Suc x)"
```

This definition uses the Isabelle meta-logic operators ⟦ and ⟧ for the premise. These semantic brackets act as pseudo conjunction for assumptions and are based on the meta implication $\Longrightarrow$. $[\![A; B]\!] \Longrightarrow P$ stands for $A \Longrightarrow (B \Longrightarrow C)$ and can be read as $A \wedge B \longrightarrow C$.

Using the more common notation for inference rules, the rules can also be written like this:

$$\textsc{EvenZero}\quad \mathit{Even\ Zero} \qquad\qquad \textsc{EvenStep}\quad \frac{\mathit{Even}\ x}{\mathit{Even}\ \mathit{Suc}\ (\mathit{Suc}\ x)}$$

[3] A set is closed if the set contains all its limits.

This notation will be used for rule-based systems, i.e. type systems, reduction relations and other inductive definitions throughout the thesis. For other formal statements, a mathematical notation is used in most instances. In some parts, it is required for the understanding to present the underlying Isabelle code; in these instances the code and constructs used will be explained as required.

Isabelle/HOL automatically generates basic induction and case distinction rules for inductive definitions on which advanced proofs can be built. To perform proofs on the structures defined by the constructs above, Isabelle/HOL offers a number of tactics. These tactics include the application rules of classical reasoning on one hand and powerful automatic proof tools on the other. Features not explained in this section will be introduced upon their use. As an example, consider this proof that the successor of an uneven number is even, made using Isabelle's Isar system:

```
lemma SucEven:"¬even x ⟹ even (Suc x)"
proof (induct x)
  case 0
  from prems have False by (simp add: EvenZero)
  thus ?case by simp
next
  case (Suc x)
  thus ?case
  proof (cases "¬not even x")
    case True
    from prems have "even (Suc x)" by  simp
    hence "False" using prems by simp
    thus "even (Suc (Suc x))" by simp
  next
    case False
    hence "even x" by simp
    thus ?thesis by (rule EvenStep)
  qed
qed
```

The proof uses induction over the variable "x" as its core principle; the initial case – zero not being even – can be resolved by contradiction. The "Suc" case requires another case distinction: If the original value is even, then the assumption leads to a contradiction, otherwise the `EvenStep` rule is satisfied. An older style of Isabelle syntax would use the command apply, as shown below.

```
apply (induct x)
apply (simp)
...
```

The two proof styles can be mixed and are generally equivalent, with readability and re-usability favoring the newer Isar style.

For modular reasoning [Kammüller, 1999], Isabelle/HOL supports locales – proof contexts for theories, that allow the instantiation in various scenarios. Locales, along the polymorphic type system allow the convenient re-use of formalizations and proofs.

Coq

Coq [Bertot and Casteran, 2004] is a frequently used theorem prover especially popular in the language meta-theory community. The logic used in Coq differs significantly from the one used in Isabelle/HOL. Instead of using a classical logic like Isabelle/HOL, which is rooted in the Russell-Whitehead logic, Coq is based on the Calculus of Inductive Constructions. The Calculus of Inductive Constructions is an application of the Curry-Howard correspondence and is based on the work of the philosophers Per Martin-Löf, Thierry Coquand and Gérard Huet [Martin-Löf, 1984; Coquand and Huet, 1988]. The Curry-Howard correspondence is founded on the observation that proofs can be seen as functional programs, where the return type of the function is the property to be proven. This means that proofs are executable and correspond to λ terms [Howard, 1969; Curry and Feys, 1958].

This approach yields a constructive style of reasoning, where the user has to prove the correctness of a statement by showing its type-correctness. The idea of this constructive, intuitionistic logic is to unify programming and verification [Martin-Löf, 1984]. The constructive logic results in some major divergences from the established logic that can be confusing for novice users. For instance, a number of formulae that are clearly true in classical logics, cannot be shown in Coq [Bertot and Casteran, 2004]. As an example, the simple formula $((P \to Q) \to P) \to P$ cannot be shown in Coq's logic, despite its truth in classical logic. See Figure 2.3 for a truth table showing the formula's truth in classical logic.

P	Q	$P \to Q$	$(P \to Q) \to P$	$((P \to Q) \to P) \to P$
f	f	t	f	t
f	t	t	f	t
t	f	f	t	t
t	t	t	t	t

Figure 2.3: Truth table for $((P \to Q) \to P) \to P$ – this is known as Peirce's formula. Following Tarski, a proposition is true iff it is true for all values of P and Q.

This is not a flaw in Coq or its logic, it is merely a side effect of using constructive logic. The infinite universe of this logic does not allow proofs by contradiction, as it is rooted in a philosophy allowing more than just either true or false. In fact, the *Tertium non datur*: – $P \vee \neg P$ is not provable directly in intuitionistic logics, like Coq's.

Using appropriate wrappers, the intuitionistic nature can be ignored or worked around in most day-to-day situations. For instance, *tertium non datur* can be proven as $\neg(\neg P \wedge \neg\neg P)$, without allowing deduction of the original notation from that[4].

The main advantage of Coq's style of declarative, constructive proofs, is that executable code can be extracted far more easily than in Isabelle/HOL[5] for example. Proofs in Coq are formulated in the language Gallina, an imperative programming language, which is also executable . Another frequently mentioned advantage is the power of Coq's type system. As a necessary result of using the Curry-Howard isomorphism, Coq's type system supports dependent types – required to express quantification – making it more expressive than the simple type system used in HOL-style provers in this regard [Kammüller, 2006]. Despite including dependent types, the type system of Coq is still decidable. The dependent types also serve as a natural module concept for Coq, presenting a very elegant approach for modular proofs.

Ignoring these obvious differences, the Coq system's abilities are close to those found in Isabelle/HOL. For instance, the inductive definition of even numbers could be written like this in Coq[6]:

```
Inductive even:NaturalNumber ⟶ Bool
where
  even_zero: even Zero
  even_step ∀ x:NaturalNumber . even x ⟹ even Suc (Suc x)"
```

This is very similar to the notation one would use in Isabelle/HOL. The general support of advanced tactics and proof methods used to lag behind Isabelle/HOL, but has caught up in recent years. All of this makes Coq a remarkable system that provides a basis for language meta theory comparable to Isabelle/HOL.

Summarizing this chapter, we presented Aspect Orientation with its concepts of aspects as pointcuts and advice, joinpoints and the notion of obliv-

[4]The inability to show $P \vee \neg P$ from $\neg(\neg P \wedge \neg\neg P)$ is because intuitionistic logic does not accept $\neg\neg P \longrightarrow P$. $P \vee \neg P$ is not provable for the same reason: Without a witness indicating whether P or $\neg P$ was true, the proposition is not valid. Note that tertium non datur is an axiom in classical logic.

[5]It should be noted that Isabelle/HOL's ability to generate code has been extended recently.

[6]In fact, the introduction of Inductive definitions marked the move from the Calculus of Constructions to its specialization, the Calculus of inductive Constructions.

iousness. We then described three core calculi for Object Orientation before finally giving an introduction into interactive theorem provers with an emphasis on Isabelle and Coq. In this Chapter, we gave a concise introduction into the key concepts used in this thesis: Aspect Orientation, Core Calculi and interactive theorem provers. We created this background in the style of a mini-survey to motivate our choice of tools – especially on the topics of the right formalism to use as a foundation and which theorem prover to select. We limited the theorem prover presentation to Isabelle/HOL and Coq, recognizing these two as the major tools for verifying language theory at the moment.

CHAPTER 3

Calculi for Aspect Orientation

A number of formal approaches to Aspect Orientation has been published by other researchers. At the time of this writing, none of the other approaches was realized in an interactive theorem prover. This section will introduce the other approaches and weigh their individual properties.

In recent years, several groups have been developing core calculi to study AOP. Their approaches extend over a broad swath of different styles, basic languages and goals. Within the field, there are approaches to analyze basic problems of language theory, including semantics, real-time behavior and types. A particularly important distinction is the notion of joinpoints, meaning the points in the base calculus at which advice can be applied. The surveyed approaches can be classified into two groups: Those using explicit labels marking these points and those using an existing construct – usually methods or functions.

This section provides a brief survey of the state of current research and the particular problems addressed by the various calculi. The approaches are listed with their aim, base calculus, the nature of their aspect-aware features and their notion of types. Section 3.1 lists the approaches based on the λ calculus, Section 3.2 the approaches based on the ς calculus. Finally, Section 3.3 lists the calculi using Featherweight Java. A condensed version of the survey can be found in Table 3.1. The chapter closes with a discussion of the approaches in Section 3.4.

Name	Basic Calculus	Type System	Joinpoints	Focus
MinAML	λ calculus	simple types	Labels	Type-preserving
Harmless Advice	MiniAML	simple types	Labels	non–interference
ς_{ASP}	simple ς	untyped	Method names	–
MINIMAO	imperative FJ	nominal types	Method names	Ownership
AspectGFJ	FJ	nominal types	Method names	Generics
Tiny Aspects	λ / ML	simple Types	Method names	Modular Reasoning
StrongAspectJ	Featherweight Java	simple Types	Method names	Types

Table 3.1: Comparison of related approaches

3.1 Lambda Calculus Based Approaches

MiniAML Core Calculus

Ligatti et al. introduced a small aspect-oriented core language [Ligatti et al., 2006] for their aspect-oriented version of the functional programming language ML. To prove the type safety of this language, a small calculus with a type-preserving translation from the language MinAML was introduced. The model is able to make a case for the type soundness in a real AOP language. Furthermore, the easily understandable calculus, based on the λ calculus, makes it well suited for type-theoretic analysis. A interesting detail is the absence of *obliviousness* in the core calculus[1], which greatly enhances reasoning about the type system.

Aim The MiniAML system is an aspect-oriented approach explicitly intended for the type-theoretic analysis of Aspect Orientation. It consists of an high-level functional aspect-oriented language and a basic calculus. The aim of this approach is the ability to reason about types in aspect -oriented programs.

[1] The language "on-top" – minAML – is oblivious.

Basic Calculus The basic calculus is based on a typed λ calculus. It realizes aspects using explicit labels for the joinpoints, arguably removing obliviousness. Aspects are represented as pairs of pointcuts and (advice-) terms. Static pointcuts are simply denoted as sets of aspect labels. Advanced pointcuts are supported via a stack construction in the evaluation environment. Aspects are denoted within the same term, meaning that there is no aspect construct setting aspects and base program apart. However, aspects and labels can be added dynamically. The evaluation of aspects is part of the operational semantics, using a recursive helper function. The return from an aspect can jump inside the term, which models the behavior of `proceed`.

Type System The MiniAML core type system is based on the simply typed λ calculus with string, boolean, vectors and integer as basic types. Moreover the system types aspects and labels. The typing function for labels enforces that the labeled subterm has the same type as the label, where the label's type is recorded in the typing environment. As mentioned before, aspects in miniAML are tuples from pointcuts and advice. The pointcut is a set of labels, required to be from the same type; the advice is a function. The typing rule for aspects requires the advice's result to be of the same type as the labels in the pointcut, under the condition that the advice's argument is of the same type. The MiniAML core supports no subtyping.

Weaving Weaving is part of the core calculus' operational semantics and uses a helper function. To weave, the helper function utilizes repeated substitution, traversing the list of aspects from the execution environment. Advice from aspects with matching pointcuts will be combined with the result from the prior aspect (or the base term). In absence of (further) aspects, it returns the identity.

Example $label.x \rightarrow x + 5 >> label\langle 3 \rangle$
Sample term for the MinAML core calculus. An aspect increasing every expression labeled with "label" by five is woven to a base term.

Harmless Advice

Harmless Advice [Dantas and Walker, 2006] is one of the first approaches for *security-typed aspects*. It uses a simple core calculus to show that *non-interference* for aspects can be enforced by a static type system. While the information–flow related features in the calculus are not very expressive, they sufficiently demonstrate the approach's novel offering of not only a sound, but a secure approach to aspects.

Aim Harmless Advice show that a certain class of security aspects can be formulated in conformance with *non-interference*. Furthermore, it shows that a simple type system is sufficient to enforce such a policy.

Basic Calculus The approach uses a *pseudo-imperative* variant of the Mini-AML Core Calculus. The aspect semantics were simplified and several language constructs – such as advice ordering – removed. A new operator to combine expressions was introduced, giving it the appearance of an imperative language, although a purely functional structure was maintained.

Type System The type system used by harmless advice is foundationally similar to the one used in the miniAML core calculus. It is enriched through the introduction of protection domains, an universal unit type and a reference type. The major addition is that the type system presented for Harmless Advice encodes non-interference statically for aspects. It thus guarantees the integrity of the security domains.

Weaving Aspects are woven in a fashion similar to that used by the mini-AML core calculus. A function is used in a reduction rule to apply advice.

Tiny Aspects

Tiny Aspects are a concept for modular reasoning with aspects. Proposed by Aldritch [Aldrich, 2005], they offer a high–level language for modules with aspects building upon a strict definition of modular advice. *Modular* in this context describes the ability to compose aspects independently and to perform an analysis of isolated aspects without a given base application. The concept is composed of the core calculus named Tiny Aspects, which was surveyed, and the larger concept OpenModules.

Aim Open Modules and Tiny Aspect are designed to allow modular reasoning about advice.

Basic Calculus Tiny Aspects are modeled after ML and thus ultimately the λ calculus. Tiny Aspects are considerably larger than other core calculi and supports the proceed statement directly. They also support named functions, resulting in structured terms. Only call pointcuts are supported, but other semantics are listed as future work.

Type System The type system is largely identical to System $F_{<:}$ and does not handle objects explicitly.

Weaving Aspects are directly applied on base functions, necessitating a large number of rules in the reduction relation.

3.2 Sigma Calculus Based Approach

Parametrized Object Calculus with Aspects

$\varsigma_{asp}(M)$ is an early example of an aspect-oriented core calculus. Written by Clifton, Leavens and Wand [Clifton et al., 2003], it is a parametric calculus allowing the study of various aspect-oriented approaches. The authors offer a number of examples of their application of $\varsigma_{asp}(M)$ to existing languages. This same group continued their research with MiniMAO.

Aim $\varsigma_{asp}(M)$ is a flexible core calculus. The instantiation of the parameter M is intended for mappings of various pointcut semantics; examples include DemeterJ; AspectJ and HyperJ. $\varsigma_{asp}(M)$ is especially designed to offer a detailed representation of proceed semantics, i.e. the traversal from aspect to base code. "Naked" ς methods are the realization of advice, "naked" meaning terms with an unbound self-parameter and no surrounding object.

Basic Calculus $\varsigma_{asp}(M)$ is an extension of the simple, untyped functional ς calculus. It includes call and execution semantics as well as a new `proceed` keyword in the operational semantics.

Type System $\varsigma_{asp}(M)$ has no new type system and is even untyped in the normal use. For the simulation of type-based pointcuts, the language uses a simple structural type system using the signature of objects as their type.

Weaving An extended operational semantics handles weaving directly, including several pointcut semantics. This adds many reduction rules to the language.

3.3 Featherweight Java Based Approaches

MiniMAO

MiniMAO can be best described as *Featherweight AspectJ*, as it extends FJ to a calculus expressively designed for the study of aspect-oriented semantics in AspectJ-like languages. As the product of ongoing and comprehensive research, it allows formalization in a notation closely resembling the accepted aspect notation of AspectJ.

Aim MiniMAO is designed to aid the understanding of aspect semantics in the accepted form of AspectJ. Moreover it serves as a formal basis

for extensions to AspectJ and AspectJ-like languages. It focuses on aspect heap effects and the semantics of `proceed`. More specifically, the calculus serves as formal basis for the AspectJ extension MAO (**M**odular **A**spects with **O**wnership).

Basic Calculus $MiniMAO_0$ serves as the basic calculus for the later incarnations of $MiniMAO$: $MiniMAO_1$, a calculus for AspectJ and $MiniMAO_3$[2], which adds ownership types and control flow effects. It was written as an extension to Featherweight Java, introducing imperative execution and assignments, allowing the study of heap effects. The calculus also adds explicit Aspect instantiation and ownership domain declarations; most other new constructs are introduced via annotations.

Type System MiniMAO extends the type system of Featherweight Java, which in turn resembles a subset of Java. The extended type system was shown to be sound in paper-proofs. $MiniMAO_3$ introduces an ownership-aware type system for aspects, which can guarantee some security constraints. The soundness proof was extended to encompass that variant. Ownership types limit the access to data to the "owner" of that data, enforcing a structure on the referencing of objects.

Weaving As with $\varsigma_{asp}(M)$, weaving is entirely handled on the level of the operational semantics. The resulting semantics are – compared to Featherweight Java – fairly large.

Aspect FGJ

Jagadeesan et al. created an aspect-oriented version of the generic extension for Featherweight Java [Jagadeesan et al., 2006]. It uses a simple Aspect Orientated feature set, limited to `execution` pointcuts without negation. However, it supports generics.

Aim The authors developed their calculus with the challenges and benefits encountered when combining generic polymorphism with Aspect Orientation in mind. They show shortcomings in AspectJ regarding types and code reuse. For instance, they observe that normal AspectJ requires code duplication to deal with generics, in opposition to modularizing crosscutting concerns in particular or reusable modules in general. Another example is the impossibility of aspect type safety in case of redefined return types due to covariance.

[2] We are not aware of a $MiniMAO_2$.

Basic Calculus The calculus used is based on the pure – functional – Featherweight Java with generics. The generics were extended to support non erasure-based handling of generics as well as the traditional erasure. On the other hand, casts were removed from the language. Aspect-oriented features are AspectJ-like, but limited to execution pointcuts without negation and dynamics. Pointcuts use method names with a small negation-free logic language.

Type System The type system is a modified version of the Featherweight Java type system. The main extension is the ability to redefine methods in subclasses, the main reduction the removal of casts. Aspects are typed, using constrained base types.

Weaving Aspect FGJ programs are statically transformed into FJ programs. Advice application is reduced to method invocation.

StrongAspectJ

The motivation for strong AspectJ [De Fraine et al., 2008] is directly related to this work: Observing obvious issues with the way aspects are handled, the group presented a calculus to prove a solution to pressing problems. The calculus differs from the one presented in this work as it is not mechanized and does not attempt to present a general theory of objects, but rather focuses on problematic situations in AspectJ.

Aim The concept is intended to formulate a type safe subset of AspectJ, solving contravariance issues and providing a future development for AspectJ.

Basic Calculus The basic calculus is an extended variant of Featherweight Java.

Type System The type system is simple, with limited subtypes using invariance.

Weaving Aspects are directly applied to base functions, leading to rather many rules in the reduction relation.

3.4 Discussion

AspectFGJ's basic approach is remarkable. Using simple, functional calculus, the authors successfully uncovered fundamental issues with aspect-oriented type systems, introduced valid generics for aspects and indicated

solutions for the aforementioned issues. Considering the significance of the problems exposed with AspectFGJ, it seems that small core calculi can extend the state of the art even without venturing into the analysis of reference semantics.

The formalization of joinpoints has a very notable effect on the whole calculus. As stated before, the survey analyzed approaches using explicit labels and approaches using existing names to identify points at which aspects can be applied. The approaches using labels tended to use leaner formalizations for aspects and weaving, as they quantify over labeled terms without indirection over names. However, it is often argued that such an approach is not oblivious, breaching the Friedman-Filman definition of Aspect Orientation. $MiniMAO_3$ constitutes the most elaborate existing approach and – with the possible exception of *Harmless Advice* – the approach most closely related to the goals of the course of research proposed in this text. However, we believe that the use of pattern-pointcuts hinders modular reasoning. Also, we have observed that a modular approach like MiniMAO cannot statically enforce the absence of control-flow influences when it also supports aspects with control flow effects. A small thought-experiment shall serve to add weight to that point: Advice can in itself expose joinpoints. Other aspects can include pointcuts capturing those joinpoints, resulting in the invocation of their advice. This holds true for aspects with ownership; aspects not following ownerships can be triggered on the execution of ownership-aware aspects. This removes any security guarantees made. Clifton et al also write:

> Advice marked with `@surround` has no control effects. [...] (Note that this allows extra joinpoints to be introduced, both within advice and the advised code)

and

> Of course, other advice that is not control-limited can cause control effects that occur at join-points within control-limited advice.

We argue that the introduction of new joinpoints in a formalism allowing control-flow affecting advice constitutes a change to the control flow. This is because control-flow affecting advice can be woven in at those – introduced – joinpoints, invalidating the static and modular results. The authors of $MiniMAO_3$ are aware of this limitation, but do not consider it substantial, an assessment we would contradict. Thus, we believe that the approach of using explicit labels is a fundamental part of any meaningful formalization. At the very least, control-flow limitations cannot be optional. In our analysis, miniMAO was found to be a well executed, but ultimately limited approach. Results gained from reasoning about a module cannot be used to

make guarantees about the behavior of the system. Even for a given system, the security constraints cannot be determined using miniMAO's approach for reasoning alone. Thus, the question has to be asked whether the price for the notion of obliviousness used in this approach might be too high.

At the other end of the spectrum is the Aspect Calculus proposed by Ligatti et al. Its scope and maturity reach that of MiniMAO and it also offers more controllable means to reason about joinpoints by featuring explicit labels. This approach had arguably the biggest influence on our formalizations, as it also focuses more visibly on type theory and boasts a very lean and – with the possible exception of the advanced stack construction – intuitive formalism. Especially the presence of a type-conserving translation to a fully-fledged language sets this calculus apart. The offspring theory of Harmless Advice shows how such formalisms can be applied to security type systems without losing the clarity of the original concept.

Aldrich's TinyAspects are similar, but more clearly designed to allow modular reasoning. They also use the concept of basing the new language proposal on a small calculus. Tiny Aspects maintains obliviousness, unlike the calculus by Ligatti et al. However, it does not support dynamic pointcuts.

Jagadeesan et al's Aspect FGJ is a subtly different case, as it lays out the challenges and benefits of having generics in AOP and is not an in-depth study of AOP itself. It also encompasses many problems specific to Java, as it has a focus on generics in Featherweight Java and thus Java.

StrongAspectJ can be seen as an in-between between Aspect FGJ and MiniMAO. It closely follows AspectJ and Java, with the stated aim to provide a strong type system for AspectJ. Unlike Aspect FGJ, the approach does not offer a solution for contra-variance issues, but merely removes them from the language. This is to closely reflect AspectJ and the idea of providing a strong type system for that language, not a new language. Also, the work is solely intended for AspectJ, not a general notion of aspects and as such not directly comparable to this work. Moreover, strongAspectJ was never mechanized in a theorem prover. Its type safety has only been proved for a subset of the language.

In conclusion, the approaches can be grouped into calculi that abstract from actual languages by using a formal calculus as basis and those using subsets of real-world languages. The calculi following the syntax of actual languages – for instance MiniMAO – show a notable overhead used for naming and accommodating concepts which we do not consider particularly interesting, for instance lexicographic pointcuts. The formal and abstract approaches, especially the MinAML core calculus, are sufficient to express scenarios like typed pointcuts elegantly and without an excessive overhead. Harmless Advice shows that such an approach can be extended without invalidating the base. Thus, we consider the abstract approaches better suited

for the study of AOP. They are able to express complicated features with a far smaller formalization, which facilitates mechanization.

CHAPTER 4

Formalizing and Mechanizing Languages

With the fundamental concepts established in previous chapters, this chapter now focuses on the inner details and requirements of a language formalization. To do so, it will first introduce the basic concepts for the syntax and semantics of programming languages before then switching to the more technical problems collected in the POPLmark challenge [Aydemir et al., 2007].

The formalization of a language, quite predictably, means constructing the language as a formal system [Curry and Feys, 1958]. This entails the definition of basic objects, the construction of a formal frame of operations and finally the development and proving of supporting theorems.

Before continuing to introduce the various incarnations of the ς_{Asc} family of aspects, it seems prudent to present the main issues encountered when formalizing a language in general and mechanizing it in a theorem prover in particular. Some of these concepts are very well known, like operational semantics and their small and big steps incarnations versus denotational semantics. Others, like α conversion and the associated problem of formalizing binders are almost unknown outside the language meta-theory community. In the meta-theory community, the search for an ideal solution has led to the issuing of the POPLmark challenge [Aydemir et al., 2007], a challenge to provide innovative solutions for the mechanization of the calculus $F_{<:}$.

The same challenge also re-iterates the benefits of developing a calculus inside a theorem prover. Complexities and nuances of proofs about languages are so subtle that it is near to impossible to achieve a fault-less result on paper, and even more so to convince others that the formalization is without

flaw. Using a theorem prover leads to the desired guarantee of 100% correctness and re-usability at a steep prize of introducing additional steps and slow progress, resulting in several times more work.

There are two different styles of mechanizing a concept: Deep and shallow embedding. A shallow embedding uses the logic of the theorem prover itself. That is, functions in the embedding are directly mapped to functions and other constructs in the prover itself. By contrast, a deep embedding entails formalizing every single detail on its own, using a level beyond the theorem prover's own.

This section shall provide a small introduction into these concepts. It serves a dual purpose, foremost to explain the choices that were made in the past years, and secondly to equip the reader with the knowledge necessary to understand the formalization presented in Part II of this thesis in detail. This chapter is largely independent of a particular theorem prover, but concentrates on the realization in Isabelle/HOL.

4.1 Syntax

The most basic part of formalizing a language is defining its syntax, i.e. the constructs that form terms and thus encode their grammar. Taking the λ calculus as example, the primitive syntax can be described like this, using three different primitives:

$$\begin{array}{lll} Lam =_{def} & \lambda\, x.\, Lam & \text{(Abstraction)} \\ & \mid x & \text{(Variable)} \\ & \mid (Lam\, Lam) & \text{(Application)} \end{array}$$

Such definitions correlate to context free grammars, normally expressed using the Bachus-Naur form [Knuth, 1964]. In the context of this thesis, the different formalisms for expressing the syntax of programming languages can be ignored, as the features and syntax of the theorem prover used dictates the type of expression used. In the case of Coq and Isabelle, datatype definitions are the tool of choice for the primitive syntax of a language. Thus, we omit the various forms and styles of grammars and their implications and merely note that datatypes, i.e. simple polymorphic and recursive sum types, are a suitable form for expressing the syntax of a calculus [Paulson, 1990].

4.2 Semantics

Beyond the raw syntax, it remains to define the *meaning* of the syntactic building blocks. This meaning is given to the syntax by defining the se-

mantics. Today, two styles of semantics are frequently used: Operational semantics and denotational semantics [h. L. Ong, 1995].

Operational semantics are a simple concept of expressing the semantics of programming languages [Plotkin, 1981, 2004; Winskel, 1993]. Formally, operational semantics correspond to inductive definitions, i.e. they are the least set closed under a set of inductive rules. This is in effect a description of how the program would be executed on a virtual machine. There are two flavors of operational semantics: Big step and small step. In small step semantics, one reduction step in the premise is related to one reduction step in the conclusion of each reduction rule. By contrast, complete reductions up to final configurations can be used in big step semantics. Generally, big step semantics relate better to to compilers, but are inadequate for proofs concerning termination [Leroy and Grall, 2009]. The main advantage of operational semantics is that their inductive nature is beneficial for performing proofs, especially proofs for type soundness [Wright and Felleisen, 1992].

The other frequent style are denotational semantics, rooted in domain theory, heavily relying on fixed points and least partial orders. The concept is that the language is described by a mathematical object, mapping terms to (simpler) terms. Denotational semantics [Reynolds, 1998; Winskel, 1993] were originally introduced by Strachey and Scott to replace operational semantics and form a higher level of abstraction than operational semantics. Thanks to their mathematical elegance that allowed compositionality and a natural handling of recursion, the style of denotational semantics quickly gained dominance. In recent times, denotational semantics became less frequently used than operational semantics, as their style is less suited for mechanization and type systems [Wright and Felleisen, 1992].

In the field of mechanized calculi, denotational semantics are an uncommon style due to their abstract nature. An operational semantic can be used as a basis for an interpreter, as the structure is directly linked to the evaluation on a simple machine. Such tools can even be automatically extracted from formalizations in modern theorem provers. These tools also close the gap between the styles, as they automatically generate mathematical objects for operational semantics. In a work like this one, the support for type soundness proofs is more important than compositionality, making operational semantics the natural choice.

4.3 Type Systems

Type theory, especially the theory of simple types, going back to Russell [Russell, 1908] and Church [Church, 1940], is a formal method to limit a calculus to a subset of well behaving terms. Well behaving usually means

that a term "does not go wrong", a concept well–suited for programming languages. The application of Russell's type theory to the λ calculus marks the beginning of the use of types for calculi and programming languages, offering a system to eliminate situations that, if allowed, could render a system unsound [Barendregt, 1992; Church, 1940]. The original invention of the type theory by Russell was triggered by the observation called "Russell's paradox". This describes situations were negated self-referring can be used to construct a paradox situation. The classical example is *the set containing all sets that are not members of themselves* [van Heijenoort, 1967, Russell (1902): Letter to Frege, page 126]. Type theory resolves the issue by restricting self-reflection [Russell, 1908].

Types systems, usually written as sets of logical rules, can encode properties. For a given term, the abidance to a type system can be enough to decide whether a given property holds – if the term is well typed, the type system *sound* and the property part of the system's guarantees. The most basic such property is the guarantee that "well-typed programs do not get stuck", which can be strengthened to "*Well-typed terms cannot go wrong*". That can encompass varying definitions of *wrong*, for instance no information flow happens from *high* to *low* in programs well-typed with regards to a type system enforcing non-interference[Sabelfeld and Myers, 2003]. The formal expression to indicate a term's typing is as shown below.

$$E \vdash t : T$$

This expression reads[1] "In the environment E, the term t has the type T".

A term being well typed does not provide any guarantees unless the type system is sound. In a sound type system, it is impossible for a term to contradict the type system's guarantees.

The established approach to prove type soundness is based on Wright and Felleisen [Wright and Felleisen, 1992]. The idea is to establish two key properties: Preservation[2] and Progress. The former – Preservation – states that if a term t of type T reduces to a term t', then t' also is well-typed. In the strict interpretation, it states that reduction does not change a term's type. Formally, Preservation reads

$$t \rightarrow_\beta t' \wedge \vdash t : T \Rightarrow \vdash t' : T$$

The other property – Progress – states that typed terms reduce or are values. For example:

$$\vdash t : T \Rightarrow \ \exists t'.t \rightarrow_\beta t' \vee \ value(t)$$

[1] The $\vdash$symbol is taken from Frege's *Begriffsschrift* [Frege, 1879]. It signifies a judgment $a \vdash b$, and reads b is provable from a.

[2] Also known as Subject Reduction.

The two parts act together in an inductive fashion, guaranteeing that nothing can go *wrong*. This proof approach is especially well-suited for small-step operational semantics and has played a major role in their renaissance [Plotkin, 2004].

4.4 The Binder Problem and its Solutions

The binder problem is so fundamental that it is regularly overlooked when reasoning about a language. The binding of variables is the most basic property of languages like the λ calculus, extending to almost any calculus in use today. Binders are the concept that gives a variable its role in a term. Examples for binder is $\lambda x.\ x$ – the variable x is bound by the binder λ.

Formalizing binders in a way beneficial for reasoning is one of the fundamental problems in language theory. It is the cornerstone of the well established POPLmark challenge. The problem[3] revolves around the concept of α conversion. Logically, meaning structurally and semantically, the two terms $\lambda x.\ x$ and $\lambda y.\ y$ – in this example the identity function – are completely equivalent. However, the variables bound by the binder in each term have different names, meaning that classical equality can not catch the actual equality of the terms. However, α conversion, one of the three basic conversion relations, states that any bound variable can be renamed.

To actually reason about a calculus, it is desirable to have a representation able to decide where terms differ in the variable names only, i.e. are equivalent up to α conversion. In order to maintain the ability to reason, a formalization has to keep the problem space small. Thus, α conversion and issues arising from the handling of binders are mostly ignored in pencil and paper proofs, despite being a fairly common source of errors [Kleene, 1962]. Regardless of the question regarding its correctness, this laissez-faire attitude is not an option in interactive theorem provers.

There are several commonly used concepts [Aydemir et al., 2007] to handle binders and α equivalence, we will present those that were considered: De Bruijn Indices, Nominal Logic and Locally Nameless Variables.

One concept that we cannot present in detail here is the use of Higher Order Abstract Syntax (HOAS) [Pfenning and Elliott, 1988; Ciaffaglione et al., 2003] [4]. HOAS uses a higher order representation of the studied language – called object language – to abstract over the binder problem altogether. That is, the binders of the logical framework are used for the object language in the form of a shallow embedding. This provides a high level of abstraction

[3] Another commonly mentioned problem is the possible ambiguity of variable names. This can be overcome by clear precedence of binding.

[4] Strictly speaking, de Bruijn indices are an implementation of HOAS.

that removes much of the complexity associated with binders. This approach seems very promising. While we feel that the concept's existence is worth mentioning, we do not consider it suitable for this thesis.

We do not mention the theoretically possible explicit handling of names – manually proving α equivalence and the Barendregt condition, as the overhead imposed on proofs is prohibitive and in most cases beyond the technically feasible. This is thus by no means a complete list, but a selection of the approaches considered feasible in the state of the art research.

De Bruijn Indices

De Bruijn Indices [Bruijn, 1972] are probably the oldest established solution for representing variables, as well as the most commonly used. They go back to the automath project [Bruijn, 1991], one of the earliest forays into computerized proofs. The basic idea is very straightforward: Instead of using names to identify variables, the distance to their binder is used. In essence, the name of the variable is replaced with its place in the structure of the term, completely eliminating explicit names and thus α conversion. To illustrate the concept, consider the following term:

$$\lambda x.\ \lambda y.\ xy$$

As one can see, it applies the second abstraction's parameter to that of the first. An equivalent term would be as follows:

$$\lambda y.\ \lambda x.\ yx$$

The beauty of de Bruijn indices is that the two notations both yield the same representation:

$$\lambda.\ \lambda.\ \$1\$0$$

In the de Bruin representation, the variable bound by the outer λ abstraction (x in the first, y in the second term) is replaced by the distance to its binder, which is 1 – there is one more binder (λ) between the variable and its binder. The second variable is bound in turn by the abstraction directly in front, with no further abstractions. Hence, it yields the index 0. Also note that the binder, the abstraction, no longer mentions the name of the variable bound; the indices encode that information in the variable identifier itself.

This solution to α conversion by removing names is the simplest initial approach for handling the binder issue. There are two major downsides of using this solution. The first is that the removal of variable names makes it harder to read the resulting terms. While named variables can be traced easily, it can also be very difficult to keep track of a variable replaced by its index. The following – tiny – example will show this issue:

$$\lambda x.(\lambda y.xy)x$$

which translates to:

$$\lambda.(\lambda.\$1\$0)\$0$$

The example is simple: The outer abstraction's variable is used as the argument of the inner abstraction's, which in turn applies the outer abstraction's variable on its argument. $\lambda x.\ xx$ would be an equivalent notation. In the de Bruijn representation, it is hard to see that the two instances of \$0 do not refer to the same variable, but that, in fact, the inner \$1 and the outer \$0 do.

The other problem with this approach is that the simple basics enforce a rather complex handling of terms. As the structure of a term changes in the process of its evaluation, the indices have to be updated. This is largely marginal in the case of the λ calculus, but quickly leads to problems in larger examples. Namely, an operation " lifting" is used to increase the indices of free variables when required. The readability of this process is further harmed by the very concept of de Bruijn indices: There is no visible difference between bound and free variables; just a – seemingly – arbitrary threshold with bound variables under and free variables above it. Handling lifting is, as problems go, not difficult in nature, but sometimes counter-intuitive and it introduces a major overhead in the formalization of many advanced theories.

The use of de Bruijn indices has another notable side-effect: It is possible to express terms that are not expressible in the normal λ calculus, for instance λ. \$4. There is only one abstraction, so any variable index greater than \$0 is meaningless, including the index \$4. On the flipside, it is also impossible to express a direct counterpart to the term $\lambda x.\ y$ using de Bruijn indices. Thankfully, these cases do not correspond to meaningful λ terms and can be ignored assuming a simple notion of well formedness.

Nominal Logic

Nominal logic is a relatively new approach for reasoning with named variables, spearheaded by Pitts [2003]. This approach has been implemented in an extension [Urban, 2008] for the Isabelle/HOL theorem prover. The advantage of using nominal techniques is the ability to express the basic lemmas about variables in a notation almost identical to the one used for the λ calculus on paper. Variables can be treated almost naturally, without introducing an unexpected abstraction. The nominal package provides the required infrastructure for reasoning in the form of nominal datatypes. Nominal datatypes realize α equivalence classes, so that the rules for equations like $\lambda x.\ x = \lambda y.\ y$ are available.

The downside to the approach is in part the very novelty of the concept: The extension of Isabelle/HOL for nominal logic has not yet reached the

maturity of Isabelle/HOL itself and is – despite undeniably fast progress – not yet established to the point where it supports the full feature set of Isabelle [Berghofer and Urban, 2007]. Notable items missing from the nominal package are the ability to use functions as parts of datatypes and – even more importantly – the code generation facilities.

The latter limitation essentially removes the nominal approach – Isabelle's being the only advanced implementation – from the consideration, especially as it is generally possible to find a bijection between terms in nominal and de Bruijn representation.

Locally Nameless

The increasingly popular Locally Nameless [Aydemir et al., 2008] [Charguéraud, 2009] approach builds upon the idea of using indices for bound variables and names for free variables. Locally nameless, incidentally already mentioned by de Bruijn [Bruijn, 1972] personally, encodes that idea into a dynamic binder concept. As mentioned before, free variables in a term are represented as names, much like variable names in classical formalizations. On the other hand, binders are used just as they are with de Bruijn indices: They do not indicate the name of the variable they bind. The seemingly paradox situation of having named variables but binders oblivious of the names of "their" variables can be resolved by having a look at the mechanics of the locally nameless representation.

To understand the intent of using the locally nameless approach, it is important to first understand that names are not used to allow the reasoning about human-readable terms. It is more the case that the approach is intended to provide human-readable proofs, i.e. it does not lead to prettier terms, as a term in a formalization using a locally nameless variable representation is virtually indiscernible from one in a de Bruijn indices based formalization. The actual motivation in using the locally nameless approach is to simplify the internal bookkeeping in the case of substitution, i.e. to avoid the constantly changing variable identifiers throughout the reduction of a term using de Bruijn indices. In fact, the solution does not require "index juggling", but provides variable identifiers that remain constant under substitution. As a result, locally nameless offers a solution for α conversion, variable shadowing and the complexity of de Bruijn indices, but not one for "pretty" variable names.

Towards that end, the locally nameless approach is footed on a few assumptions. The first assumption is that there are always "fresh" variable names, i.e. names that are not used by any other free variable in the same term. With this in mind it is possible to introduce the two operations unique to the locally nameless approach: The opening and closing of terms.

Opening opens a term for analysis; it introduces free variables for a given binder's bound variables. Formally it takes a variable index i, a name x and a term t. The operation, $\{0 \mapsto x\}t$ – also written t^x, as the initial value of i is 0 – is defined as replacing a bound variable i with a free variable of the name x. An important restriction to this is that x may not occur in t, as that would cause an incorrect capture. For every binder that is traversed, i is incremented by one. In the following example, $\$i$ indicates a bound variable with the index i, while free variables are marked only by their names:

$$
\begin{aligned}
&(\lambda.\lambda.\ \$0\ \$1)^x \\
=&\{0 \mapsto x\}(\lambda.\ \lambda.\ \$0\ \$1) \\
=&(\lambda.\ \{1 \mapsto x\}\lambda.\ \$0\ \$1) \\
=&\lambda.\ \lambda.\ \$0\ x
\end{aligned}
$$

Opening replaces the variable bound by the outer abstraction with the free variable x. Note that the operation does not introduce a name for the variable bound to the outer abstraction, i.e. it remains as $\lambda.$ instead of the $\lambda x.$ that might have been expected. The resulting term is no longer *closed*. Closing is, under the assumption that opening used a fresh variable, the reciprocal action of opening, replacing free, named variables again with bound variables identified by structural indices.

The question arises, stated intent aside, where the benefits of locally nameless materialize and furthermore how the system's variables are supposed to work at all. The concept of indices is well established and hardly new for most people who had some exposure to the internals of language theory. The names in the locally nameless representation are a different kind of concept and deceivingly similar to classical variable names. This question has to be answered by research, whether there is a real benefit in using the locally nameless representation.

4.5 Induction

Induction as the main method for performing proofs about structural formalizations is the second big challenge particular to mechanizing languages. Its role is to provide an infrastructure for performing proofs on the formalization. It might seem surprising to find such a well established principle in this compilation, but induction is an essential part of formalizing languages and the POPLmark challenge [Aydemir et al., 2007].

As languages are usually defined as sets of inductive definitions and datatypes, induction is the tool of choice to use in proofs. For induction

to work properly, an induction schema has to be provided, ideally without explicit user input by the proof assistant. However, this is not as trivial as it may seem, as language formalizations tend to employ novel approaches, not necessarily covered by the automatic mechanisms for the generation of induction schemas. Common examples where induction can pose an issue are the aforementioned binders; providing and maintaining the generation of induction schemes is one of the challenges requiring a formalization of binders not directly recognizable to people used to paper proofs. However, even in relatively conservative extensions, it can become very challenging to derive meaningful induction schemes from the basics provided by the proof system. For example, we use more than ten different induction schemes in our formalization, which requires finding the right induction for each proof – a step that is both error-prone and time consuming.

4.6 Code Generation

The final step in a language mechanization is the ability to extract code from the formalization. This code promises complete correctness regarding the properties proven on the formalization. Being obviously specific to the domain of interactive theorem proving – or at least computer aided reasoning – this marks the step where classical formalizations and mechanizations diverge completely. It is also a step that is frequently seen as the biggest reason for using interactive theorem provers in language development in particular and formal systems in general.

Not just because of that importance, but certainly accelerated by it, the field of code extraction is important to the communities around provers. Some theorem provers are natively aided by their logic in this extraction process, notably Coq, due to its underlying calculus of inductive constructions. Others, notably Isabelle/HOL, employ a transformation algorithm that generates executable code from the constructive elements of definitions and proofs in proven transformation steps [Berghofer et al., 2009]. Nonetheless, it can be argued that the code generation abilities of Coq are superior to those found in Isabelle/HOL, solely based on the merits of using a constructive logic.

Regardless of the proof assistant used, the challenge to yield a usable program from a mechanization is non-trivial. This is due to technicalities on the one hand: It is generally only possible to yield executable code from a subset of the syntactic abilities provided by a theorem prover. A non-constructive proof in Coq is not suitable for code-generation, as is the unbounded use of quantifiers or nominal logic in Isabelle/HOL. In both languages, significant steps are being taken to provide the code generation for more constructs

[Berghofer et al., 2009] while also improving the performance [Lochbihler, 2009] and usability of the generated code.

Even with all of the advances, code generation usually leaves a bridge to gap. The formalization – and mechanization – calls for certain deviations from the pure paper-based calculus and logical simplifications. Examples are the removal of variable and method names in favor of numeric names or even de Bruijn indices. Other examples for deviations from paper proofs encompass changes in syntax and encoding to ease the manipulation in the proof environment. Regardless of the various pressing reasons of these changes, the code generated will include them as well, which is usually not intended. While numbers can serve as names for an educated reader willing to invest the work, this cannot be expected of "normal" users. Even more, sometimes the generated objects are not printable or take a shape not even remotely similar to the original formalization on paper. To overcome these cosmetic issues, further work is required to make the results of code generation usable – either in the proof assistant itself to add the required bridges for input and output, or as manual shell around the generated code.

The practical use of code generation in language design is the extraction of prototypes for interpreters and type checkers. In the broader field of theorem proving, code generation is starting to evolve into a discipline of software development, where the programs are entirely developed in a formal environment. The recent presentation of a compiler for a subset of C realized entirely in Coq [Leroy, 2009] is a popular showcase item for that trend.

Summarizing this chapter, we gave a list of six parts required for formalizing a language in a theorem prover. The first half of these were classical and certainly known to the vast majority of users: The essential syntax and semantics are a core concept across a large number of fields. The second half presented topics of a nature more specific to mechanized language theory. The handling of binders rarely is of importance outside of language theory, but is a fundamental consideration when designing a calculus, especially a mechanized calculus. The topic of induction is natural for a calculus, but rarely important for a practical handling of languages, the final topic of code generation is solely important for mechanized approaches.

This part presented the context for the remainder of the thesis: The tools to be used, the problems to solve, the related work and the vocabulary of the domains. More specifically, we gave an introduction to Aspect Orientation and the preliminaries of language (meta–)theory in interactive theorem provers. These preliminaries are the building blocks from which we assemble our aspect-oriented core calculi in the following part.

Part II

ς_{Asc} – Its Syntax, Semantics and Mechanization

CHAPTER 5

An Untyped Calculus with Aspects

This chapter introduces the first version of the mechanized aspect calculus. The following chapters will extend the calculus to address the fundamental issues of modularity, typing and subtyping in the design of aspect-oriented languages.

To present the design the AOP calculus, a number of basic design fundamentals must be first established. The foremost step for the calculus concerns the formal foundation to use. There are three principal calculi that are commonly used for the formal study of advanced language concepts. The oldest and most widely known is the λ calculus [Barendregt, 1984]. While the λ calculus is certainly a viable foundation for almost any scenario, it is very cumbersome to embed Object Orientation into the calculus. Considering that the aim of this thesis is the study of aspects in an object-oriented context, it is desirable to use a native object-oriented calculus instead.

There are two established calculi for reasoning about object-oriented scenarios: The ς calculus [Abadi and Cardelli, 1998] and Featherweight Java [Igarashi et al., 1999]. Either of these calculi is object-oriented and thus well-suited for the scenario at hand. The ς calculus is very small, a trait beneficial for reasoning, especially when extending a calculus. Its structural type system allows for a very direct reasoning about types. By contrast, Featherweight Java is a functional subset of Java, i.e. every term in Featherweight Java is also a Java program. This introduces a rather cumbersome nominal type system into the calculus. Even more importantly, by using a subset of Java, possible results lose generality, as they are linked to Java. This thesis is intended to cover a broader base than just Java based lan-

guages. As a result, the much more versatile ς calculus was used.. The different calculi are presented in greater detail in Section 2.2.

The next important consideration is the style of the aspect extension. The survey in Section 3 already showed that two basic approaches exist: The use of introducing explicit labels to abstract over joinpoints on the one hand, and the approach of using method names on the other. The latter approach is aimed at achieving a close relation with AspectJ, but adds a layer of complexity and ambiguity, which is prohibitive for strict reasoning. This is due to names being not necessarily unique. The former approach abstracts from the actual pointcuts by assuming labels to be present, greatly simplifying the reasoning at the cost of strict obliviousness. For this work that aims at establishing a strictly proven base for aspects, the former approach was taken, as it can simulate the latter without excessively increasing the complexity of the model.

This first step of adding aspects to the ς calculus, presented in this chapter, is based on the simple, untyped ς calculus. The aspect extension ς_{ASC} is straightforwardly implemented on the very lean ς calculus.

5.1 Formalization of the Basic Calculus

The formalization of the new calculus follows a very natural design. Our basic concept is to use the established ς calculus as a basis and adding just as much as is required. This has the benefit of allowing us to rely on the known properties of the calculus and thus focus on the added features. We execute the mechanization as a conservative extension, i.e. it does not introduce any axioms and does not alter the internal logics of Isabelle/HOL. Moreover is designed in the style of a deep embedding [Boulton et al., 1992], i.e. the calculus is entirely formalized in Isabelle/HOL, but does not rely on the logic of Isabelle/HOL itself. For instance, the substitution is defined as a function on terms and does not use Isabelle's substitution.

The new calculus shares the basic syntax of the ς calculus, but is enriched by a new constructor for the representation of labels. These labels mark joinpoints in the calculus, i.e. places where aspects can act. The next section will introduce this concept in detail. Bear in mind that the ς calculus is entirely functional in its nature. The basic syntax for the new, basic calculus can be expressed as shown below.

$$
\begin{array}{rll}
a, b ::= & x & \text{(Variable)} \\
& [m_i = \varsigma(x)b_i] & \text{(Object, } l_i \text{ distinct, } i \in 1..n\text{)} \\
& a.m & \text{(Method Invocation)} \\
& a.m \Leftarrow \varsigma(x)b & \text{(Method Update)} \\
& l\langle a \rangle & \text{(Aspect Label)}
\end{array}
$$

The mechanization of the basic object construct is particularly challenging and requires some preliminary theories. Objects in the ς calculus are lists of labeled methods written as $[l_i = \varsigma(x)b_i]^{i \in 1..n}$. The mathematical construction most closely resembling this idea is a map, a simple function mapping names to values – or in this particular case, names to methods. However, objects have a finite number of methods and thus only values – methods – for some names. Isabelle/HOL's logic – HOL – is a logic of complete functions, i.e. it is not possible to express functions in general and methods in particular that are only defined for some names from the domain of possible names. To overcome this, a technical workaround is used: Instead of mapping names to methods directly, names are mapped to an `option` type. This option type has two constructors: `None` and `Some x`. The former constructor signifies that that the map has no entry for the name, i.e. that there is no value defined for the key supplied. Conversely, the latter constructor signifies that there is a value defined for the name in the map and it wraps that value. To access the wrapped value, the function `the` is used: the (Some x) = x. This technical encoding satisfies the totality requirement, as the map itself is defined for any name given as input – the option type abstracts over the partiality. Another important function in this regard is the domain function `dom`, which yields the set of names for which a value is defined in the map.

Initially, maps might look like a perfect fit for the representation of objects. Their built-in option workaround allows a good representation of the concept of ς objects in Isabelle/HOL. For the mechanization however, the solution is not yet satisfactory. The domain of names is infinite and not generally inductive. Nonetheless, as described in Section 4.5, it is desirable to use induction as proof principle for language theory. Thus, imposing an inductive structure over the map type is a requirement for using maps in the mechanization. We use a novel concept to resolve this impasse: Finite Maps. Our finite maps are based on the map construction, but differ in an important detail: They are defined on a finite domain of names. This use of a finite carrier domain allows us to derive an induction scheme for the finite maps, based on the induction over finite sets. The general principle is very straightforward: If a condition holds true for the empty map and after adding an entry to a finite map, then the proposition is proven.

Using these finite maps, the syntax is expressed as a datatype In Isabelle/HOL:

Definition Syntax of ς_{Asc}

```
datatype sterm =
    Var nat
  | Obj "Label -~> sterm" type
```

```
| Call sterm Label
| Upd sterm Label sterm
| Asp_Label nat sterm  ("_⟨ _⟩")
```

This mechanization corresponds to the classic syntax, but has a few subtle alterations. Foremost is the addition of the `Asp_Label` constructor to accommodate our concept of aspects. In addition, the technical aspects of mechanizing the calculus require a number of further changes to the calculus, such as the inclusion of finite maps.

The very first constructor, `Var` for variables uses a natural number instead of a name or similar approach. This is due to the use of de Bruijn indices for the representation of variables. Using de Bruijn indices, as introduced in Section 4.4, means that variables are identified by the distance to their binder. This completely abstracts from names, removing α conversion completely. At the same time, it leads to a somewhat complicated definition of substitution, that will be introduced later in this section. The next significant difference is the shape of the constructor for objects. The notable change – the addition of a type annotation needed later – does not change the primitive semantics and will be explained when we introduce typing in the next chapter. The non-presence of a ς notation in the syntax is a design decision of the calculus; method bodies are not discernible from other terms, removing one constructor.

The object constructor uses the finite maps introduced above. It uses a finite map for lining method implementations to their names, allowing us to maintain induction. Note that all objects share the same finite carrier domain of possible method names. The remaining constructors are direct representations of their ς equivalents.

Isabelle/HOL generates induction, case distinction theorems and establishes fixpoints for datatypes automatically, which in most cases are sufficient for most tasks. However, in our case, the use of finite maps in the datatype required the manual definition of an induction scheme, as the scheme provided by Isabelle/HOL was not practical. This step alone would have been less viable with normal maps, as the usable induction scheme is built on the one for finite maps, which in turn relies on the well-foundedness of finite maps. Automatically generated schemes would create the cases solely for the `Some y` and `None` cases of the map, not taking the finiteness of the datatype into account. By using the advantages of the finite maps, we were able to derive a schema that uses the much more intuitive cases $f(\emptyset)$ and $[\![(Pf) \wedge l \notin dom\ f]\!] \Rightarrow P(f(l \mapsto t))$, similar to the well-known induction on lists which uses cases for empty list and the appending of one more element.

The use of de Bruijn indices is a decision that was neither easy nor uncontested and has been revisited countless times throughout the development.

Advantages of de Bruijn indices are the maturity of the concept and the sheer simplicity of dispensing with names completely. Moreover, a translation from terms using named variables to one using de Bruijn indices is entirely feasible [Bruijn, 1972]. Since the early stages of the project, we have been investing considerable effort in examining the various solutions to the binder problem (see Section 4.4). One of the most promising concepts was the Isabelle/HOL extension for nominal logic [Urban, 2009], then not part of the Isabelle distribution and highly experimental. After correspondence with the author of the nominal package, it was concluded that the use of functions – finite maps – in the datatype was beyond the abilities of the nominal logic package. As a result, de Bruijn indices were adopted as the principal representation for binders and used for the remainder of the project.

Our realization of the de Bruijn indices is very straightforward. Two situations act as binders, the first being the `Obj` constructor, the second being the argument of the update – i.e. the new method body. In either case, the current self object is represented by the index '0'. The substitution on ς_{Asc} terms is defined as a primitive recursive function as shown below:

```
primrec
  subst        :: "[sterm, sterm, nat] ⇒ sterm"  ("_[_'/_]")
and
  subst_option :: "[nat, sterm, sterm option] ⇒ sterm option"
where
  "(Var i)[s/k]     = (if k < i then Var (i - 1)
                           else if i = k then s else Var i)"
 |"(Call a l)[s/k]  = (Call (a[s/k]) l)"
 |"(Upd a l b)[s/k] = (Upd (a[s/k]) l (b[(lift s 0)/(k+1)]))"
 |"(Obj f T)[s/k]   = let
                      (t = λ l.(subst_option (k+1) (lift s 0) (f l)))
                      in (Obj t T)"
 |"(asp_label⟨ t⟩)[s/k] = asp_la⟨(t[s/k])⟩"

 |"subst_option n s None     = None"
 |"subst_option n s (Some t) = Some (t[s/n])"
```

Informally described, $t[s/k]$ replaces all occurrences of the variable with the index k with s in t. The formal definition is, as shown above, slightly more complicated for two reasons. The first is the nature of the datatype we used to model the objects. Objects use a finite map, which is a partial function, while HOL is a logic of total functions. To overcome this, the partiality of the map is encoded by assuming that the map is a total function, pointing to an option datatype. This means that for parameters in the domain of the map, the result is `Some value`, where `value` is of the type `sterm`. For parameters outside the domain, the result is `None`. To handle this, the substitution function uses a λ expression to roll the function application over all methods

of an object. In turn, this λ application needs a mutually recursive function with the substitution proper to handle these option values. The option cases are handled by the `subst_Some` and `subst_None` bodies. `None` is skipped, as nothing cannot contain variables. In the case of `Some`, the function is applied to the payload.

The second reason for the substitution being somewhat more complicated than one would expect are the de Bruijn indices. Upon substitution, the indices in the terms have to be adjusted to meet the structure of the term. This restructuring involves the reduction of all variables greater than one to be substituted by one, indicating that the binder was removed as we use substitution as invocation. Another factor is that the index to be substituted gets increased at each binder passed, reflecting the nature of the de Bruijn indices. Adjusting variables in the terms to be substituted, is the second, more complicated part of index handling[1]. It is realized in a function called `lift`, which is defined as shown below:

```
primrec
  lift        ::"[sterm, nat]        ⇒ sterm"
and
  lift_option ::"[nat, sterm option] ⇒ sterm option"
where
  "lift (Var i) k     = (if i < k then Var i else Var (i + 1))"
 |"lift (Call a l) k  = Call (lift a k) l"
 |"lift (Upd a l b) k = Upd (lift a k) l (lift b (k + 1))"
 |"lift (Obj f T) k   = Obj (λ l. (lift_option (k + 1) (f l))) T"
 |"(lift (i ⟨t⟩) k    = (i ⟨(lift t k)⟩)"
 |"lift_option k None     = None"
 |"lift_option k (Some t) = Some (lift t k)"
```

Lifting is applied to binders upon substitution, or more directly, whenever the substitution operator moves within the scope of another binder. This is to keep the indices in the term to be substituted consistent, as one variable can – and will – have different indices depending on the position of its usage in the term. Informally, lifting adapts all variables in the term to be substituted to the structure, i.e. by increasing variables by one for each traversed binder. That is, a variable that has the index 0 at the initial root level has the index 1 behind a binder, 2 behind another binder and so on. Consider this example:

```
Obj ( foo ↦ ς Var 1)[bar/0]
```

The variable 0 is to be replaced by the term *bar*. The substitution operator passes the binder Obj, resulting in

[1] Juggling comes to mind.

```
Obj ( foo ↦ ς Var 1[bar/1])
```

Which yields the expected:

```
Obj ( foo ↦ ς bar)
```

The variable 0 at the initial level is identical with the variable 1 inside the binder Obj and has to be adjusted accordingly. Now, consider this slightly different example:

```
Obj ( foo ↦ ς Var 1)[Var 0/0]
```

The substitution substitutes the variable 0 with the variable 0, i.e. the correct expectation is for the substitution to have no effect. Now the role of lifting becomes obvious.

```
Obj ( foo ↦ ς Var 1[Var 1/1])
```

When passing the binder, the term to be substituted was lifted, incrementing the variable count, which again yields the expected result:

```
Obj ( foo ↦ ς Var 1)
```

This only affects the variables that are bound at the level of the initial binder; variables belonging to binders within the term are unaffected. As we use de Bruijn indices, the barrier between bound and unbound variables is solely marked by a numeric threshold and not obvious from looking at the variables themselves. An important fact to remember is that the self variable has the index 0 inside a method's body, as the substitution happens on the body instead of the object.

With substitution defined it is possible to continue to the next step and introduce the operational semantics for the basic calculus. Following Plotkin and Felleisen [Plotkin, 1981; Wright and Felleisen, 1992], we use small step operational semantics, as these ease later type-related proofs and also lend themselves naturally for mechanization. In Isabelle/HOL the rules are written in a slightly different, but similar notation. The fundamental Beta rule shall serve as an example:

$$[\![\ \texttt{l} \in \textbf{dom}\ \ \texttt{f}]\!] \Longrightarrow \texttt{Call (Obj f T) l} \rightarrow_{Asc} \texttt{the(f l)[(Obj f T)/0]}$$

From here on, notation throughout this thesis follows the standard notation for inference rules. Axioms, i.e. rules without premise, are shown without a bar. The form is:

$$\textsc{Example}\ \frac{\mathit{premise}}{\mathit{conclusion}} \qquad\qquad \textsc{Axiom}\ \mathit{axiom}$$

The rules defining the reduction relation in Figure 5.1 use this notation. See the Appendix A for the original Isabelle/HOL version. To introduce the

$$\textsc{Beta}\ \frac{l \in \mathbf{dom}\ \ f}{\texttt{Call}\ (\texttt{Obj}\ f\ T)\ l \rightarrow_{Asc} the(f\ l)[(\texttt{Obj}\ f\ T)/0]}$$

$$\textsc{Upd}\ \frac{l \in \mathbf{dom}\ \ f}{\texttt{Upd}\ (\texttt{Obj}\ f\ T)\ l\ a \rightarrow_{Asc} \texttt{Obj}\ (f\ (l{\mapsto}a))\ T} \qquad \textsc{SelL}\ \frac{s \rightarrow_{Asc} t}{\texttt{Call}\ s\ l \rightarrow_{Asc} \texttt{Call}\ t\ l}$$

$$\textsc{UpdL}\ \frac{s \rightarrow_{Asc} t}{\texttt{Upd}\ s\ l\ u \rightarrow_{Asc} \texttt{Upd}\ t\ l\ u} \qquad \textsc{UpdR}\ \frac{s \rightarrow_{Asc} t}{\texttt{Upd}\ u\ l\ s \rightarrow_{Asc} \texttt{Upd}\ u\ l\ t}$$

$$\textsc{Obj}\ \frac{s \rightarrow_{Asc} t \qquad l \in \mathbf{dom}\ \ f}{\texttt{Obj}\ (f(l \mapsto s))\ T \rightarrow_{Asc} \texttt{Obj}\ (f(l{\mapsto}t))\ T} \qquad \textsc{Asp}\ \frac{s \rightarrow_{Asc} t}{l\langle s\rangle \rightarrow_{Asc} l\langle t\rangle}$$

Figure 5.1: The inductive definition of the reduction relation.

general principle of these rules, it is important to remember the finite maps mentioned above and in particular the encoding of partial functions by using the option type. The operator `the` selects an element in the option datatype when it is defined, i.e. unequal to `None`. Similarly, the function **dom** yields the domain of a (finite) map, i.e. all arguments which are not mapped to `None`.

As the next step, the relation itself is realized as an inductive predicate, which encodes the least closed set that satisfies the rules. These rules can be read as: If the precondition is met, then the two terms in the conclusion is valid. In our case, all conclusions have the form $s \rightarrow_\beta t$, meaning $(s,t) \in \beta$, which means s reduces to t in one step. The complete definition is depicted in Figure 5.1.

Most importantly, the first rule – `beta` – encodes the very core of the calculus, as it expresses method invocation. In this rule, the substitution `[(Obj f T)/0]` replaces the self parameter for the outermost variable – which always has the index 0 – in the object's field labeled with l – `f l`. This is the general mechanism for method invocation in our calculus. In other words, the invocation of a method is performed by replacing the self parameter of

the called method – which always has the index 0 – with the enclosing object. This principle to represent function application by substitution is, in the following section, used again for weaving aspects. Remember: Due to the functional nature of the calculus, there is no difference between a copy of the term and the actual object. Thus, the rule for update simply replaces the old term with the new one and yields the resulting object. Context reduction is realized in the next four rules: `selL`, `updL`, `updR` and `obj`. These cases add the reduction of subterms inside objects and statements. Finally, the rule Asp adds the groundwork for our conservative extension to the core calculus. It allows the reduction inside labeled terms.

Throughout this thesis, we use the notation $s \rightarrow_{Asc} t$ to indicate that the term s reduces to the term t in one step. A derivative relation of the single-step reduction is the transitive, reflexive closure of the normal reduction relation, $s \rightarrow^*_{Asc} t$. This notation indicates that s and t are either identical or that s reduces to t in an unknown number of steps. $\rightarrow^*_{Asc}$ also establishes equivalence in respect to β conversion, i.e. $s \rightarrow^*_{Asc} t$ implicates that s and t are terms representing an equal value.

A formal benefit of mechanizing operational semantics in Isabelle/HOL is that the system automatically proves fixpoints and other important properties, yielding a mathematical object describing the semantics.

5.2 Adding Aspects

We extended the basic ς calculus to include aspect-oriented features, as shown in the preceding section. The main objective was to preserve the complete problem without including unnecessary detail in the formal model. As a result, we decided to use a minimalistic definition of aspects as entities consisting of one pointcut and advice.

The most important design decision regarding the aspects concerns the base program rather than the aspects themselves. Specifically, the way joinpoints are realized in a functional calculus in a fashion allowing modular reasoning is of key importance. We decided to use the approach pioneered Ligatti et al [Ligatti et al., 2006] of introducing explicit labels into the base language that identify the possible joinpoints. Whether this harms the obliviousness ideal of Aspect Orientation is a question of philosophical nature. From a pragmatic perspective, we argue that the labels can be inserted into the base program by a mechanism maintaining obliviousness. More precisely, it is not unrealistic to assume that a high-level compiler inserts the labels. Even more, labeling can be exhaustive, i.e. every possible subterm can be labeled, leading to a much finer granularity than in "real" oblivious languages. Following this train of thought, we state that labeling does not harm oblivi-

ousness in any substantial way. However, on the flip-side, labeling does allow explicit reasoning about subexpressions in a term. This ability to identify all instances where an aspect might act, even if the set of these instances is entirely abstract in nature, allows a much stricter linking of base and aspects for reasoning. Text-based patterns offer no stable base for reasoning, as they are volatile in nature and can break by simple renaming. This is why we argue that removing names from their extra semantics – just as we did it for variables – is not just valid, but even desirable[2].

Building upon the labels in the term, the realization of aspects follows naturally. We see aspects as pairs of pointcut and advice, as shown in Figure 5.2. Pointcuts are simply realized as lists of labels; if a given label is in the list, the advice applies to the expression so marked. This simple concept is decidable, allows the application of a given aspect several times and abstracts completely from the original base program. The concept for advice is similarly simple: A function written in the calculus itself is the core of the concept. Normal methods in ς_{Asc} use Var 0 to refer to their direct self-variable, encoded as de Bruijn index. Aspects, or more precisely advice, use the $Var\ 0$ as their *base* variable. Weaving then invokes the aspect – if the aspect is applicable – just as method invocation would. The base variable is replaced with the term marked by the label; the result replaces the original marked term. There are three dimensions to the consideration of

$$\overbrace{\{\underbrace{[lab_1, lab_2, ...]}_{Pointcut};\ \underbrace{\lambda x.fx}_{Advice}\}}^{Aspect}$$

Figure 5.2: A ς_{Asc} aspect

this concept. The most obvious question concerns the removal of labels. As labels have no semantics in the primitive reduction relation, they will stop the evaluation of a program. Thus, labels must be removed after weaving is complete. This leads to the second question: When is weaving completed? Removing labels during weaving is not an option, as more than one aspect might be woven to a label or a single aspect more than once. Hence, removal must occur in a dedicated step after the completion of all weaving. This is performed by a *delabel* operation, which is part of the extended semantics of the calculus. The final question evolves around the traversal of terms. An aspect might be applicable within itself or deeper inside the joinpoint already under consideration. Hence, the weaving operation has to traverse

[2] As stated before, we do not think that this reduces the expressive power of the calculus. We could have used the method labels as names for pointcuts, but that would have limited the calculus. Using labels introduces a finer granularity of possible joinpoints and a less volatile way of reasoning. Labeling all methods according to their name would be trivial.

the complete term. This means that it may not stop when reaching a label, but continue the weaving inside the labeled subterm. As traversed labels remain in place, they have to be removed during the reduction.

Our simple formalization of weaving, as shown below, has a number of advantages. It is easy to understand and formalize, yet powerful enough to simulate different approaches to aspect orientation. This simplicity is made possible by the approach of using explicit joinpoint labels, i.e. the approach of assuming that pointcuts were already resolved by another tool.

```
primrec
  weave        :: "[ sterm, aspect ]        ⇒ sterm"
and
  weave_option :: "[ sterm option, aspect ] ⇒ sterm option"
where
   "weave (Var n) a      = Var n"
   |"weave (Call s l) a  = Call (weave s a) l"
   |"weave (Upd s l t) a = Upd (weave s a) l t"
   |"weave (Obj f B) a   = Obj (λ  l. (weave_option (f l) a)) B"
   |"weave (l ⟨ t ⟩ ) a  = (if  (l mem  (pc a)) then
                               (l ⟨ (adv a)[(weave t a)/0] ⟩ )
                            else
                               (l ⟨ (weave t a) ⟩ ))
   |"weave_option None a     = None"
   |"weave_option (Some t) a = Some (weave t a)"
```

5.3 Properties of the Untyped Calculus

Confluence

Even the untyped core calculus has a number of remarkable properties that simplify the analysis. An important example is the confluence of the base calculus' reduction relation, i.e. the determinism of the evaluation. A graphical representation of the Church-Rosser property is given in Figure 5.3.

Theorem 5.3.1 (Confluence of the Reduction)
$s \rightarrow^*_{Asc} t_1 \wedge s \rightarrow^*_{Asc} t_2 \Longrightarrow \exists t'.t_1 \rightarrow^*_{Asc} t' \wedge t_2 \rightarrow^*_{Asc} t'$

The proof of the Church-Rosser property re-uses parts of a proof framework for proving the Church-Rosser theorem for the λ calculus by Tobias Nipkow [Nipkow, 1996]. This framework shows confluence not by embedding the calculus in the – confluent – λ calculus, but uses instead the classical approach as per Barendregt [Barendregt, 1984].

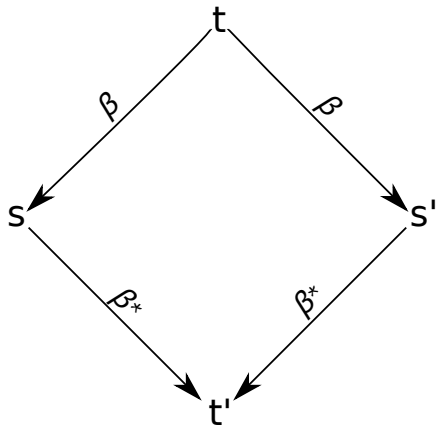

Figure 5.3: The diamond predicate.

Following the outline of Nipkow's proof framework for the λ calculus, we were able to re-use some of his theorems, most notably the diamond predicate and the final predicate shown below.

Theorem 5.3.2 (Confluence Directly)
$confluent\ r_{par} \wedge r \subseteq r_{par} \subseteq r^* \longrightarrow confluent\ r$

Using this sketch, we define a parallel semantic relation $s \Rightarrow_{Asc} t$, i.e. an operational semantic half-way between big-step and small-step for the calculus. Then we show that the new relation is greater than $\rightarrow_{Asc}$, i.e:

$$(s \rightarrow_{Asc} t) \rightarrow (s \Rightarrow_{Asc} t)$$

This means that any pair (s, t) which is part of $\rightarrow_{Asc}$ is also in $\Rightarrow_{Asc}$. The next step then establishes that $\rightarrow^*_{Asc}$ is in turn greater than the newly introduced parallel reduction: $\Rightarrow_{Asc}$, i.e.

$$(s \Rightarrow_{Asc} t) \rightarrow (s \rightarrow^*_{Asc} t)$$

In effect we thus show that the transitive closure of $\Rightarrow_{Asc}$ is identical to the transitive, reflexive closure of $\rightarrow_{Asc}$, i.e. $\rightarrow^*_{Asc}$. The complete parallel reduction relation can be seen in Figure 5.4.

The idea for using this in-between relation is the concept that it is sufficient – and much easier – to show the confluence of an arbitrary relation, if that relation is between the confluent single-step reduction relation and its transitive, reflexive closure.

Thus, we show the confluence of our parallel reduction relation, which we have shown to be larger than the single step reduction and smaller than

$$\textsc{Var}\quad Var\ n \Rightarrow_{Asc} Var\ n$$

$$\textsc{Obj}\quad \frac{dom\ s = dom\ s' \qquad \forall l{\in}\mathbf{dom}\ s.the(s\ l) \Rightarrow_{Asc} (s'\ l)}{\texttt{Obj}\ s\ B \Rightarrow_{Asc} \texttt{Obj}\ s'\ B}$$

$$\textsc{Upd}\quad \frac{s \Rightarrow_{Asc} s' \qquad t \Rightarrow_{Asc} t'}{\texttt{Upd}\ s\ l\ t \Rightarrow_{Asc} \texttt{Upd}\ s'\ l\ t'}$$

$$\textsc{Upd'}\quad \frac{\texttt{Obj}\ s\ B \Rightarrow_{Asc} \texttt{Obj}\ s'\ B \qquad t \Rightarrow_{Asc} t' \qquad l \in \mathbf{dom}\ s}{\texttt{Upd}(\texttt{Obj}\ s\ B)\ l\ t \Rightarrow_{Asc} (\texttt{Obj}\ (s'(l{\mapsto}t'))\ B)}$$

$$\textsc{Sel}\quad \frac{s \Rightarrow_{Asc} t}{\texttt{Call}\ s\ l\ \Rightarrow_{Asc} \texttt{Call}\ t\ l}$$

$$\textsc{Beta}\quad \frac{\texttt{Obj}\ f\ B \Rightarrow_{Asc} \texttt{Obj}\ f'\ B \qquad l \in \mathbf{dom}\ \ f' \qquad b \Rightarrow_{Asc} b'}{\texttt{Call}(\texttt{Obj}\ f\ B)\ l\ b \Rightarrow_{Asc} (the(f'\ l))[(\texttt{Obj}\ f'\ B), b'/0]}$$

$$\textsc{Lab}\quad \frac{s \Rightarrow_{Asc} s'}{l\langle s\rangle\ \Rightarrow_{Asc} l\langle s'\rangle}$$

Figure 5.4: The inductive definition of the parallel reduction relation.

the transitive, reflexive closure. From this result, it is possible to show the confluence of the original reduction relation β_{Asc} by using the framework.

As a result, we were able to prove the confluence of the base calculus, i.e. the label-enriched ς calculus.

Aspect Compositionality

For the aspects, we were able to derive a closely related property: A concept of aspect compositionality [Kammüller and Sudhof, 2008b]. Aspect compositionality is the answer to an important question regarding weaving. Weaving, the process of combining aspect and base code, can be implemented at different times throughout a program's lifecycle. The classical idea is compile time weaving, i.e. aspects are woven into the program by the compiler. However, newer developments are leading towards load-time or run-time weaving in order to allow the treatment of aspects and base programs as independent modules.

Our notion of compositionality answers that question: There is no difference between compile time and run-time weaving for compositional aspects. Figure 5.5 shows the concept: For a compositional aspect A, the final re-

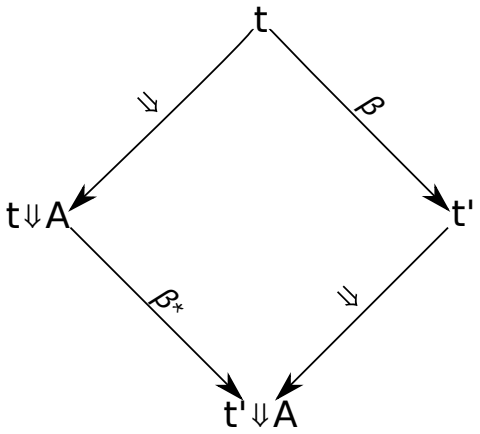

Figure 5.5: Aspect Compositionality

sult is not influenced if a term t is first reduced to t' before an aspect is introduced, or if the aspect is woven first and the reduction applied later.

The theorem showing this also hints at one minor difference. While t reduces to t' in one step, it might take more than one step for $t \Downarrow A$ to reduce to $t' \Downarrow A$. This is because the aspect might introduce additional operations that make the one-step reduction not applicable, as they require several additional steps to reduce to the same term.

Theorem 5.3.3 (Compositionality of Aspects)
$justoneFV\ A\ \wedge t \rightarrow_{Asc} t' \Longrightarrow t \Downarrow A \rightarrow^*_{Asc} t' \Downarrow A$

The predicate $justoneFV$ states that the aspect has just a single free variable. Conceptually, this free variable is the base variable used to access the advised subterm upon weaving. This is noteworthy, as the condition $justoneFV$ is not directly related to typing. However, we are able to show that any well typed aspect satisfies the condition in Chapter 6.

5.4 Expressivity of the Plain Calculus

We argue that this simple first step can simulate a large number of aspects in a modular fashion. *Modular*, in this case means that the aspects are not linked to the base program and are not written in the same term, but in a distinct entity. Our calculus achieves this, as weaving is realized using labels, with the aspects and the base program being separate prior to weaving. There are three questions to be taken into account when considering the possible aspects. The first and most obvious observation is that the calculus

is entirely functional and has no mutable state. One aspect can be woven at several places in a program, but it cannot share a global state for all these instances. Thus, the calculus is for instance unable to realize a stateful observer. We do not consider this to be a significant limitation, as the safety of an aspect does not depend on its state, but rather on the possible actions an aspect can perform on the base term. I.e. the expressive power of an aspect is the main interest of the calculus.

The second question to be asked regards the role of the labels. While the labels themselves do not influence the semantics of the base calculus, it can be seen that they carry potential semantics, these being the possible actions of an aspect. Therefore, it could be argued, that the addition of labels jeopardizes the Obliviousness property of Aspect Orientation. We reject that argument with two simple thought experiments. The idea that labels harm Obliviousness is based on the perception that labels are added at precisely the points intended for aspects. This is a limited view on the matter; for on one hand, we have to treat any place where an aspect might act regardless of the actual presence of an aspect, as we are striving to reason about Aspect Orientation – so this does not pose a distinction between a labeled and a label–less[3] approach. On the other hand, it is important to remember that there is an infinite number of label identifiers, so that it is easy to envision a function that labels every single possible joinpoint (method bodies, method invocations, ...) in the base program. Such a process would not impede on Obliviousness, as the base program is not altered in any semantic fashion. Then, another function could translate lexicographic pointcuts into label sets; there is related work using this approach in real-world scenarios [Avgustinov et al., 2007]. This shows, that the label approach is able to embed the label-less approach and thus more powerful. As a result of this thought experiment, we see no limit posed to our calculus by labels, especially considering that they provide a much cleaner theory.

The role of dynamic poincuts has to be considered as the third and final point. Dynamic pointcuts are pointcuts employing dynamic conditions, including the control flow in the program, boolean conditions and similar factors. It can be argued that our approach to reasoning, performed under the assumption that pointcuts were already resolved, does not offer a feature directly comparable to dynamic pointcuts. Avgustinov et al. have presented an approach for the static handling of aspects [Avgustinov et al., 2007] by inserting labels according to the static evaluation of AspectJ pointcuts. We believe that the considerations expressed there also apply to our calculus: It is possible to resolve a significant subset of so-called dynamic pointcuts statically. This is especially true for boolean conditions – which can be handled in the advice – and for many cases control flow conditions. Finally,

[3]Such approaches use existing method labels, so that they are rather label-reusing, than label-less.

and most importantly, we intend our calculus for reasoning about safety and security considerations. We thus do not lose expressivity by assuming that any dynamic pointcut is evaluated to be true, as we have to assume that it did already for modular safety reasoning. However, we concede that this approach to dynamic pointcuts does contradict some of the goals initially formulated for the calculus. Nonetheless, we feel that the simple calculus' ability to handle dynamic pointcuts not dependent on mutable state is a step towards safe aspects.

This first state of the calculus has no means to restrict the power of aspects, thus it is entirely possible to formulate aspects that render the term in an "stuck" state. Clearly, the main interest is in aspects that are benign, i.e. not harmful to the evaluation of the base program. Hence it is clear that extensions of the calculus have to address two questions, namely which *aspects can be rejected* and *what effects* this has on the set of expressible aspects in the calculus. The following chapter will expand on this question by introducing means to discern aspects beyond their compositionality.

In summary, we have introduced the basic formalization and mechanization of an extended untyped ς calculus in this chapter. We also introduced a minimalistic notion of aspects and established how such aspects interact with base programs in a form of weaving. As a contribution to the research in mechanized meta-theory, we have performed a case study to show how the language semantics of the ς calculus can be expressed with different variable representations. We also formally proved the Church Rosser property for the base calculus, establishing the determinism of our semantics. Finally, we discussed the implications of the calculi's design and thus laid the groundwork for the extensions of the ς_{Asc} calculus to be presented in the following chapters.

CHAPTER 6

Modular Types for Aspects

Building upon the calculus developed in the previous chapter, we now unveil our concept of typing for the base calculus and aspects. The type system presented in this chapter allows the modular typing of aspects and also guarantees strong type safety for the calculus as a whole.

While we can express aspects with the untyped approach, we have one glaring problem: There is no way to statically determine whether an aspect is safe for a given program or not. That is, a-priori, the validity of the addition of an aspect to a term is entirely unknown. This is an unsatisfactory state, calling for means to statically decide whether a given aspect can b added. A type system is a viable way of encoding a notion of correct behaviour into a language statically, i.e. all terms that are typable do not lead to certain error situations. Modularity is another key design fundamental for our type system: We believe that aspects and base programs should to be typable independently of each other. This belief is rooted in the claim that aspects are modules. It is a core requirement for modules to be independently checkable with clear error locality.

The need for a sound type system for Aspect Orientation can be easily motivated. As an example, we revisit the initial example for soundness concerns in Aspect Orientation. Figure 6 shows the example from the Introduction. The aspect asp overwrites the method `test` of type `Test` with a method of type `String`. This results – in AspectJ – in a run-time error, falsely stating the first call to `test` as the error location. By the standards applied to mainstream programming languages, this is an obvious type error that would be caught during compilation. In this case, the error is made

worse by the breached error locality – it is in an aspect, but acts in the base program – and the untyped nature of the pointcut – another method `test` of another class might work without any issue. A sound type system for aspects would prevent such situations from occurring by catching such oversights at compile time.

```
public class Test
{
  public Test test()
  {
    return this;
  }
}
 public aspect asp
{
  Object around() : call(* *.test(..))
  {
    return "oops";
  }
}
```

Figure 6.1: Example for typing issues in AspectJ.

The major problem when designing type systems is caused by the surprising complexity of the problem. Any new construct in a language can introduce unforeseen dependencies and many languages saw their type system proven inconsistent. For this first evolutionary step in the typing of the calculus, we introduce a type system in the tradition of the simply typed λ calculus, i.e. a simple first-order type system. This type system is then extended by adding the so-called "width subtyping" [Pierce, 2002].

6.1 Simple Types

The concept of a type system is the ability to check a program for execution errors statically. There are three variables in the concept, the most obvious being the concept of an execution error. What conditions constitute an error is open to interpretation, but has to be defined when creating a type system. The second such variable is the strictness of the type system. While a type system can be unsound, i.e. allow certain error conditions to manifest regardless of typing or not consistently detecting such conditions, it is clear that a rigorous approach calls for a strict and sound system. Last, but certainly not least, the power of the type system is to be considered. While a type system can guarantee the absence of execution errors, it is a

common side effect that it will reject some programs that would not have caused an error. This set of falsely rejected terms can be reduced through type systems with more flexibility, for instance by introducing concepts like subtyping. However, there is the limitation that higher order type systems are not generally decidable, i.e. using too powerful constructs can reduce the applicability. We cannot allow unsoundness in a type system intended for a formal calculus, and cannot risk undecidability if we want to maintain a connection to the practice of Aspect Orientation. Thus, a classic first order – simple – type system in the tradition of Church's λ calculus [Church, 1940] based on Russel's theory of types [Russell, 1908] is the most promising answer[Hindley, 1997].

For the ς calculus, the standard simple type system is the original type system by Abadi and Cardelli [Abadi and Cardelli, 1998]. Compared to the λ calculus, the type system is on the one hand very similar, but on the other hand much more complex. The complexity stems from the self-referencing structure of objects.

6.2 Formalization of Basic Types

The system we adapted operates without primitive types; its only type is that of object. This is a result from the ς calculi's idea that everything is an object, comparable of the notion that everything is a function in the λ calculus. As a result the type datatype uses a finite map to relate labels to types – just like the object constructor in the term datatype wraps a finite map to map labels to terms. The following recursive Isabelle/HOL datatype defines the possible types.

```
datatype type =  Object (label ↦ type)
```

The actual type of a given term is thus a nested structure that looks very much like an object. We consider this type system to be structural, as equivalence of types is solely based on equivalent structure [Pierce, 2002]. There is no condition stating that different names for types break equality[1]. This also means that there is not just a superficial similarity between objects and types. In fact, a type is structured exactly like the objects belonging to it.

At this point, we introduce a few terms for use with our types. As a type is very similar to the objects belonging to it, it is important to keep the terms used separate. Specifically, a type is conceptually a specification of the instances it describes. Types are composed of labels mapped to *features*,

[1]Note that different method names do break equality.

where the labels are the same method names used in the objects, but instead of mapping to the method body, they are mapped to the type of the method – its specification. We call a named type inside a type *feature*. All instances – i.e. all objects typable with the type – of a given type are its `members` or member instances. It is important to remember the two different directions of decomposition possible here, the set-theoretical and the structural one. We use *members* for objects belonging to a type and *features* for specification details of the type – for instance methods. The typing judgments as shown

$$\textsc{Var}\ \frac{x < |E| \qquad E_x = B}{E, L \vdash \texttt{Var}\ x : B}$$

$$\textsc{Call}\ \frac{E, L \vdash a : B \qquad l \in \mathbf{dom}\ B}{E, L \vdash \texttt{Call}\ a\ l : B_l}$$

$$\textsc{Obj}\ \frac{\mathbf{dom}\ b = \mathbf{dom}\ B \qquad \forall i \in \mathbf{dom}\ B.\ E\langle 0 : B\rangle, L \vdash b_i : B_i}{E, L \vdash \texttt{Obj}\ b\ B : B}$$

$$\textsc{Upd}\ \frac{E, L \vdash a : B \qquad l \in \mathbf{dom}\ B \qquad E\langle 0 : B\rangle, L \vdash n : B_l}{E, L \vdash \texttt{Upd}\ a\ l\ n : B}$$

$$\textsc{Lab}\ \frac{E, L \vdash a : B \qquad i < |L| \qquad L_i = B}{E, L \vdash i\langle a\rangle : B}$$

Figure 6.2: The typing judgments for the labeled ς terms. Note how L is used in the LAB rule.

in Figure 6.2 are written as an inductive definition, similar to the operational semantics given by $\rightarrow_{Asc}$ in Chapter 5. Following the usual syntax of typing judgments, the ternary typing operator $E, L \vdash t : A$ can be read as "in the environment E, L the term t has the type A".[2] Moreover, the environment can be extended by adding variable-type mappings to E with the operator $\langle\rangle$. For instance, $E\langle 0 : A\rangle, l \vdash t : B$ reads: In the environment E with the variable 0 having type A, the term t has the type B. Environments are realized as a stack, automatically increasing the index of all entries after the insertion by one, in accordance with the behavior of variable indices due to

[2] This notation is based on Frege's Begriffsschrift[Frege, 1879], but is now the generally accepted notation for typing judgements.

de Bruijn indices and scopes. The unary **dom** function returns the domain of a finite map, which is in turn a finite set.

To represent the lookup in a map, we use indices, for example B_l. I.e., for a given finite map B and a label i, B_i yields the entry in B associated with i, if there is one. The operators **dom** and lookup are overloaded to allow their application on ς types as well as on the maps wrapped by ς objects.

Based on these notations, we can continue to introduce the typing judgments. The rules are very straightforwardly interpreted as follows.

- The VAR rule for typing variables has to be read as "if the variable is bound then the expression's type is the one referenced in the variable environment for that variable" – all bound variables have types associated with them in the environment.

- The CALL rule for typing method invocations asserts that the callee is well-typed and that the method is declared in the callee's type. If so, the resulting type is the one found in the callee's type for the particular method. In this rule the similarity of objects and their types is particularly obvious.

- The OBJ rule for objects is notable, as we enforce conformity with the type annotation in the rule, linking syntax and typing. This move differs from the classic ς formalization and enables us to formalize objects in a stricter way. The actual rule states that the domains, being the notation for the existing fields/methods, of the object and its annotation have to match and that all methods of the object have to be well-typed under the assumption that the variable 0 – the *self* variable – is of the object's type.

- Similarly, the object update UPD rule asserts that the object to be updated is well typed and that the method/field to be updated is defined in its type. Moreover, the new value for the method/field has to be well typed assuming self – 0 – to be of the updated object's type.

- Unlike the reduction relation, the labels are first-class citizens in this type system. To type labels, we introduced a new environment L, which maps labels to types. LAB, the corresponding typing judgment, enforces that a given label may only be used to mark terms identically typed. Label environments are of particular importance when it comes to the well-formedness of aspects.

6.3 Typing for Aspects

We added aspects to the base calculus in Section 5.2 as a minimalistic construct to capture the problem space. These aspects are not part of the base term and are in fact modules in their own right. It is a key contribution of this thesis to present a type safety notion for aspects that maintains their modularity while still being strong enough to prove type safety for the base term. Towards this end, we already introduced the realization of aspects as a non-invasive extension to the base calculus.

We see the major feature of this approach in its simplicity and the nearly complete decoupling of aspects from the base in the definition of weaving and in the type system. This in turn allows us to treat aspects as independent modules, while still being able to guarantee static type safety. Modularity is also evident in the proofs: The proofs of type safety reduce to a theorem that shows that weaving preserves types [Kammüller and Sudhof, 2007a].

The typing of aspects is realized by a condition `wf_asp`:

Definition Aspect Well Formedness

$$\texttt{wf\_asp}\ \texttt{L}_\texttt{T}\ \langle \texttt{pc, adv} \rangle \equiv_{df} \forall\ \texttt{l} \in \texttt{pc.}\ []\langle \texttt{0: L}_\texttt{T}\texttt{!l} \rangle, \texttt{L}_\texttt{T} \vdash \texttt{adv: L}_\texttt{T}\texttt{!l}$$

It states that an aspect has to be well-typed respective to an interface `L`. More precisely, it expresses that for each label in the aspect's pointcut, the advice is of the type listed in the environment for that label, if the aspect's self parameter was of that type. The function application is indicated by assuming 0 to be of the type given by the label environment at some point `l` and thus fixing 0 in `adv` accordingly. This guarantees that an aspect's replacement of the original labeled expression conforms to the same type.

In other words, the result of weaving an advice has to conserve the type under the label. A notable benefit of this approach is that the condition is not based on a particular base-term, but is entirely based on `L`, which can be seen as generic interface. Another added benefit is the indirect instantiation of the label type using all labels in the pointcut via `L`. This allows polymorphism of aspects, as the typing is checked against a set of types.

Thus, the – for instance – application of aspects without base effects is possible on any marked term, as such advice would be typable for any base type. Moreover, as the condition `wf_asp` is part of the typing constraints, we also prove the decidability (using a primitive recursive predicate) to guarantee static aspect typability.

So far, we have described how aspects are typed using `wf_asp`, but have not introduced the system that combines the base typing and the aspect

typing. The interface L is in fact the same construct that is used for typing terms in the preceding section, as in $E, L \vdash t : A$. More precisely, the rule for typing labels relies on L being present in the type environment, as shown in the LAB rule of Figure 6.2.

This use of the environment establishes that a label is only used with subterms of a given type in the base calculus, just as it is used to limit the types an aspect is allowed to return. The two type statements are combined in a final well-formedness predicate, as shown in the definition below. The predicate states that the base program has to be typable with the empty variable environment using the same label environment $\mathtt{L_T}$ as the aspect.

Definition Compatibility of Base and Aspect Types

$$\mathtt{wf\ L_T\ \langle pc,\ adv\rangle\ t} \equiv_{df} \mathtt{wf\_asp\ L_T\ \langle pc,\ adv\rangle} \ \wedge\ \exists\ \mathtt{T.\ [],L_T \vdash t : T}$$

Using this strong definition of aspect typing, we are able to prove that weaving of well-typed aspects always yields well-typed programs. This means that both type safety theorems remain valid, when well-typed aspects are woven.

We thus are able to statically describe a strong and modular typing definition for aspects without requiring a concrete base program. More importantly, we are able to prove static type safety for our typing of aspects.

6.4 Properties of the Typed Calculus

We were able to show a number of properties for this simple type system. The first property is the uniqueness of types, which means that each term has at most one type. This is a property very helpful for reasoning, as type identity can be derived directly if a term has two types.

Theorem 6.4.1 (Uniqueness of the Type System)
$E, L \vdash t : T \wedge E, L \vdash t : T' \implies T = T'$

The more important property is type soundness. Following the Felleisen approach of defining type soundness as subject reduction and preservation, the proof splits into two theorems. The first theorem, Progress, states that any well-typed term reduces; i.e. if a term is typable, then it is either a value or it can β reduce in one step to another term. In the ς_{Asc} calculus, we consider objects to be values. An necessary addition to the theorem was added in the form of the *delabel* operation. *Delabel* is used in the theorem

to remove all aspect labels from a term, acting as the final step of weaving. Just like the labels themselves, the operation is transparent w.r.t. typing [Henrio et al., 2007]. Our approach to the proof of type soundness deviates from the original proof for the ς calculus [Abadi and Cardelli, 1998], as it does not rely on an *outcome* function.

Theorem 6.4.2 (Progress)
$E, L \vdash t : T \implies \exists t'.\ (delabel\ t) \rightarrow_{Asc} t' \vee t = (Obj\ f\ T)$

Progress is complemented by another theorem, Subject reduction. Subject reduction states that if a term reduces to another term, then that resulting term has the same type as the original.

Theorem 6.4.3 (Subject Reduction)
$E, L \vdash t : T \wedge t \rightarrow_{Asc} t' \implies E, L \vdash t' : T$

The two parts of type soundness are inter-meshing, forming an induction over the system. If a term is well-typed, then there is another term that it reduces to (Progress). As the original term reduces to another term, the reduction result has the same type as the original term, thus it is well-typed – subject reduction stays applicable, thus does progress. It is easy to see how this combination encompasses the whole system. Thus, we have formally proved the type soundness for the basic type system. While the type system is sound, it does not enforce normalization, unlike the simply-typed λ calculus. This means, that even a well-typed term can diverge.

One question remaining is the soundness of the aspect typing, which we could prove by two slight variations of the theorems shown above. The big advantage of using a construct L to separate the typing of aspects from the typing of the base application is the modularity that this approach yields. A base program can be typed just as an aspect would without any link between the two – a notion of aspects that is very much sought after in the real world. Even more importantly, we were able to extend the formally established type soundness to aspects. The important factor is the label environment L that connects aspect and base types [Kammüller and Sudhof, 2008a, 2007b].

Theorem 6.4.4 (Aspect Progress)
$wf\_adv\ L\ A \wedge E, L \vdash t : T$
$\implies \exists t'.\ t \Downarrow A \rightarrow_{Asc} t' \vee delabel(t \Downarrow A) = delabel(Obj\ f\ T)$

Theorem 6.4.5 (Aspect Subject Reduction)
$wf\_adv\ L\ A \wedge E, L \vdash t : T \wedge delabel(t \Downarrow A) \rightarrow_{Asc} t'$
$\implies E, L \vdash t' \Downarrow A : T \wedge E, L \vdash delabel(t' \Downarrow A) : T$

Both of these theorems are results of the stronger proposition that weaving well-formed aspects produces well-typed results. Together they firmly establish that weaving and our notion of aspect typing are sound.

Even more importantly, we were able to prove that well–formed aspects are compositional by showing that the type system enforces all premises for compositionality as defined in Section 5.3. Thus we are able to prove that all well-typed aspects are well behaved w.r.t. weaving. The theorem below merely pronounces the fact, as the compositionality of typed aspects can be derived from the soundness of the type system.

Theorem 6.4.6 (Well-typed Aspects are Compositional)
$wf\_asp\ L\ A\ \wedge t \rightarrow_{Asc} t' \Longrightarrow t \Downarrow A \rightarrow^*_{Asc} t' \Downarrow A$

Thus we were able to introduce a type system for Aspect Orientation that while treating aspects as individual modules, still guarantees strong type safety.

6.5 Extending the System for Simple Subtyping

As we have shown in the preceding section, types in ς_{Asc} are unique. I.e., every term has at most one type. While this is actually a useful property to have for a type system, it also is quite obvious that the expressivity of the type system does not allow any kind of polymorphism and is thus very limited.

A step to relax the strictness of the type system is the addition of subtyping, which adds the well-known "is-a" polymorphism to the calculus. Intuitively, subtyping establishes a hierarchy on types, where types lower in the hierarchy – more special – express that their members offer properties compatible with all the types higher up in the hierarchy. This hierarchy is called the subtyping relation, <:. In nominal systems, like Java, this subytping relation between two types is established by creating an explicit link between two types. By contrast, structural type systems like the one presented here, this relationship is established merely by the structure of types.

To include subtyping in the calculus, we extend ς_{Asc} to $\varsigma_{Asc<:}$ by adding basic subtyping.

Width Subtyping

The most important property when considering a subtyping system is the guarantee that any feature present in a given supertype is also present in

any subtype. To properly allow subtypes, the type system must allow subsumption, i.e. any term typable with a given type must be typable with all supertypes of that type. There are two orthogonal ideas of subtyping: Width and depth subtyping. The former uses the idea that larger types are more special, the latter that types with more special members are more special.

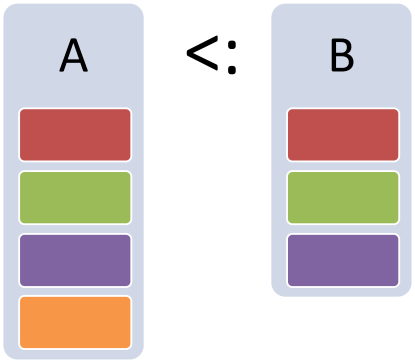

Figure 6.3: Width Subtyping: A and B are identical in all shared features, but the more special A is larger.

For straightforward subtyping, width subtyping is a viable step to achieve an easy-to-use formalization of subtypes that adds itself to the type system naturally. Figure 6.3 explains the concept of width subtyping: For two types to be in a subtype relation, they have to be identical in all their shared members, but the more special type can have more members. The subtyping relation for this style of subtyping is predictably simple, an important factor as it simplifies proofs and thus makes this approach a viable test bed for concepts with subtypes. Its sole rule, as shown in Figure 6.4, is that all type members of the more general type also have to be present in the more special type. The relation is a partial order, i.e. is transitive, reflexive and anti-symmetric. Moreover, it is easy to see that the empty type – `Object` $\emptyset$ – is the most general type. It is thus the top element of the subtyping relation and the shared supertype of all types – including itself.

$$\frac{\text{SUB}\quad \forall l \in dom(B) \qquad A_l = B_l}{A <: B}$$

Figure 6.4: The subtyping relation for width subtyping.

This subtyping relation is compatible with the type system shown in Figure 6.2 in Section 6.2, so that only very few changes were required for the typing relation. Notably, the rules needed extension to allow for a more special type than required for fulfilling the premise to be present. Example 6.5 uses the CALL rule; all other rules were changed accordingly. The premise was amended by $A <: B_l$ and the conclusion now states that the term has the type A instead of the original B_l.

$$\textsc{Call Orig}\ \frac{E, L \vdash a : B \qquad l \in \mathbf{dom}\ B}{E, L \vdash \texttt{Call}\ a\ l : B_l} \qquad \textsc{Call Width Sub}\ \frac{E, L \vdash a : B \qquad l \in \mathbf{dom}\ B \qquad B_i <: A}{E, L \vdash \texttt{Call}\ a\ l : A}$$

Figure 6.5: The original Call rule (left) and the one adapted for subtyping (right).

With this altered definition, we can prove that the system establishes subsumption.

Lemma 6.5.1 (Subsumption)
$A <: B \wedge\ e, L\ \vdash\ t\ :\ A\ \Longrightarrow e, L\ \vdash\ t\ :\ B$

The presence of the subsumption property extends the flexibility of the base calculus and the aspect typing enormously. Where the original type system required a label to match the type in the label environment exactly, it now suffices that the labeled term is typable as a subtype of the original entry. On the aspect side, an aspect can now replace the original type with a more special type, i.e. can alter the base term more drastically and can store information in extra fields. Here a major strength of the formalization comes to bear: As the aspect typing and the term typing are completely modular, the aspect type system gained the added expressivity almost without changes to itself. Merely being based on a more powerful base type system extends the capabilities.

Properties of the Calculus with Width Subtyping

Compared to the plain type system, we lose the type uniqueness property for an obvious reason: Subsumption establishes that all terms typable as T are also typable as any supertype of T. All other properties, most importantly the soundness of the type system, the aspect typing and the aspect compositionality stay valid, although requiring new proofs for all properties.

A notable effect of the relaxed type system is that it adds considerable power to well-typed aspects. Aspects can now be applied not just to la-

bels of a given type, but also to any labels having a subtype thereof. This is especially important considering the typing judgment for aspects, which states that the type under the label has to be preserved. By accepting subsumption, this automatically means that the type under the label can be specialized, i.e. the aspect can replace the original term with one of a more special type. This adds an axis of variance to the calculus, the importance of which will become obvious in the following chapter.

Summarizing this chapter, we have introduced a simple type system for the extended base calculus and mechanized said system. We then presented a modular concept for typing aspects that is able to type aspects without restricting them to a particular base program, while also guaranteeing strong safety properties. On that basis, we formally proved the soundness of the type system and of the aspect-oriented extension and weaving. We then showed that the type system establishes aspects to be in that class. In a final step, we extended the system to include a simple notion of subtyping while maintaining all important properties of the former incarnation, most importantly type soundness and the compositionality of aspects.

CHAPTER 7

Subtyping and Variance Issues

The $\varsigma_{Asc<:}$ type system removed some of the strictness of the original type system presented in Chapter 6, while maintaining type soundness and other important properties of the calculus. However, the system is not yet able to capture many of the typing issues found in real-world aspect-oriented languages. The reason for this is that the simple subtyping created by adding width subtypes is not as powerful as the subtyping found in object-oriented languages like Java: It lacks variance. This chapter introduces an extended version of the calculus, which adds the expressivity required to model situations such as the example shown below.

```
public class Point {
  public Point test() {
    return this;
  }
}
public class ColoredPoint extends Point {
  public ColoredPoint test() {
    return this;
  }
}

public aspect asp {
  Object around() : call(Point Point.test(..)) {
    return new Point();
  }
}
```

Here, the pointcut of the aspect includes the method `test` of all instances of the class `Point` and its subclasses. However, the subclass `ColoredPoint` redefines the return type of that method, which leads to a situation where advice that would be type safe for `Point` instances is no longer type safe for instances of `ColoredPoint`. This violates the golden rule of subtyping and subclassing [Pierce, 2002, Chapter 15], namely that an instance of a type is a member of all the supertypes of that given type as well. A non-uniform handling of instances of supertypes and subtypes is the result, which contradicts the modularity expected in modern languages.

To express such situations, the calculus needs a more powerful notion of subtyping, one able to capture issues like variant return types or parameter types instead of the pure width subtyping introduced in the preceding chapter. This is because – as shown in the example – type errors manifest when a subtype has a method of a different signature than the supertype, something not possible in width subtyping. In order to accommodate such re-definitions involving variance situations, we extend the system to allow depth subtyping, i.e. a concept of subtyping where related types do not vary in their size only, but also in the types of their features. The result is an extension of the calculus. This time, the extension is twofold. First, the calculus itself is extended to allow method parameters, then the type system is extended to include depth subtyping.

7.1 Variance Problems in Subtyping

The situation shown in the opening of this chapter is a classical variance issue [Pierce, 2002]. Specifically, the redefinition of a method's signature in conjunction with an aspect leads to an inconsistency in the type system, making it unsound. This is not an unprecedented development, since the early days of Object Orientation, new features were known to add variance issues.

The arguably most famous variance issue stems from the early days of class based subtyping. In the language Eiffel, it is allowed[1] to re-define method parameter types in a co-variant fashion, in line with the expectation many developers have regarding subtyping [Cook, 1989].

To explain the underlying issue, we use the established approach to show the four possible variances when re-defining a method. For theoretical purposes, we can assume a method to be a function. Based on that, the type of the method parameter[2] is the function's domain, and the method's return

[1] The typing issues were not limited to parameter types and the language still allows this. The dispute surrounding the Eiffel type system is ongoing.

[2] Multiple parameters can be seen as a vector and do not change the scenario.

type the function's range. Figure 7.1 shows the basic four variance scenarios, ignoring invariance where neither domain nor range change. The set on the left hand side contains two nested subsets; the middle subset represents the domain of the original function. A nested subset is a set more special than the original, an outer set is more general. Conversely, the range/return type of the function is expressed by the set to the right, with the outmost set being more general than the original return type and the innermost set being more special.

Compared to the original method, shown in the center, the return value can be either from a more general type (a) or from a more special type(b). The same applies to the parameter types (c)(d).

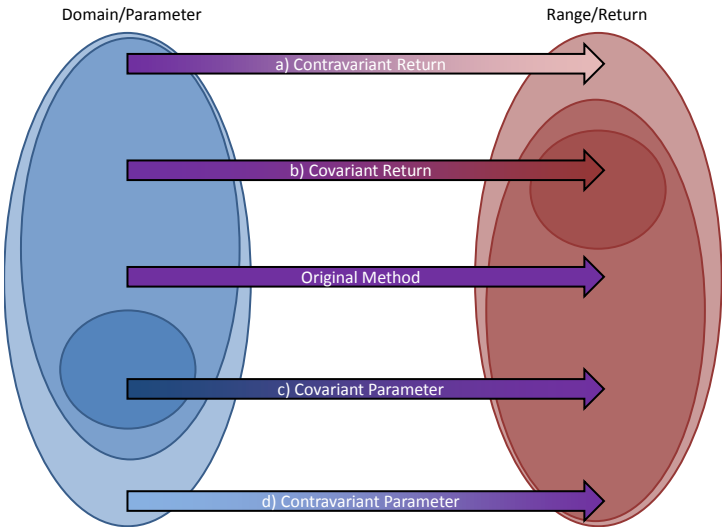

Figure 7.1: The possible variance scenarios when re-defining a method.

However, not all of the options a) to d) lead to consistent results. Remembering the rule of subtyping that any member of a subtype has to be conforming to all supertypes of that subtype, it is easy to identify the allowable cases. For a redefined method, any scenario where the original method was used has to still be valid. This means that returning a more general type (a) cannot be valid, as the more general result would not be compatible with scenarios where features of the more special original type are expected. The same applies to Eiffel's feature of allowing more special parameter types (c). As the redefined method has to accept all parameter values that are accepted by the original method, requiring a more special type does not guarantee viable results either. The other two variance situations (b)(d) lead to working programs, however, and are generally referred to as "conform". A return value can be more special, as a return value of a more special type could

still be used in any static scenario; more general parameters still allow for all the parameters that the original method accepts. Figure 7.2 summarizes the concept of a conforming re-definitions. From now on, we will refer to methods with more special return types and more general parameter types as being "more special". Conversely, a method with a more general return value and a more special parameter value is "more general". Note that the terms are selected to match the terms used for types so that a more special type is composed of more special features. Logically, it could be argued that the terms should be reversed.

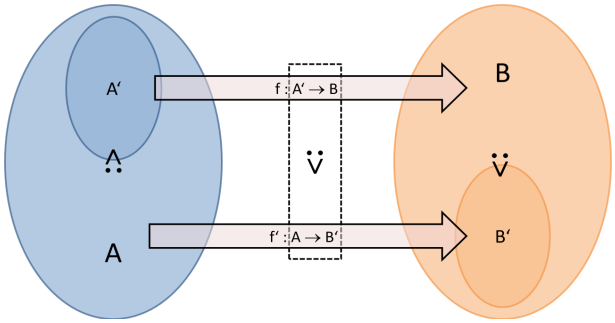

Figure 7.2: Conforming subtyping for methods.

The variance issue in Eiffel, as introduced above, is a very clear-cut example for a typing issue caused by the inclusion of variance into a type system. There, the only variance involved was the redefinition of a method, for which the conforming redefinition introduced above constitutes a clear solution. However, as shown in the motivating example, there are cases[3] where more than one concept of variance applies, leading to an unexpected variance situation.

Generally speaking, it is an accepted observation that additional degrees of freedom in a programming language's type system are not guaranteed to be compatible, regardless of their individual soundness. If two or more mechanisms allow for a variance to happen at the same point, the result can be assumed to be inconsistent, unless proven otherwise. This conservative notion stems from a pronounced lack of compositionality in typing concepts.

Specifically, aspects on their own can soundly[4] replace methods with more special methods. On the other hand, the soundness of the conforming redefinition of methods in subtyping is a well-known fact. However, the combination of these two orthogonal means of redefinition does not yield a sound result, as we have seen from the motivating example at the beginning of this Chapter, where both features were used in combination.

[3] The invariance for references of references in C++ is the result of another example.

[4] AspectJ not being type sound in that regard is not due to the impossibility of soundly solving the issue – see Section 6.5 – , but rather a design flaw in the AspectJ type system.

7.2 Extending the Calculus for Depth Subtyping

The ability to establish a subtype relation with signature changes is known as "depth subtyping". To add depth subtyping to the calculus, another limitation has to be overcome. As shown in the preceding section, the plain ς calculus is not well suited for subtyping involving the variance of method types. The reason for this is not a fault, but its very simplicity: The plain calculus, as used up to now, does not support method parameters. Thus, some extended re-engineering is required to introduce method parameters, so that these parameters can be affected by variance.

The ς calculus, and thus $\varsigma_{Asc<:}$, does not discriminate between methods and fields of an object. Both can be equally written to and the passing of parameters is entirely handled by the means of storing the parameter values in a field first. Considering that the type problems in aspect-oriented languages often are founded in incorrect handling of method parameter types, it is desirable to formalize such parameters in the calculus. Furthermore, the type system as presented in Chapter 6, restricts the access on fields of the same object, meaning that it is still very hard to pass parameters to functions – width subtyping relaxed the system to a degree, but not enough to express method passing naturally. To be able to reason about variance issues in the calculus, we introduce a method parameter into the basic calculus. This is an established step for enriching ς calculi, which allows reasoning about method parameters.

In Isabelle/HOL, the extension of the variables to accommodate parameters was formalized by introducing a new kind of variable, splitting the variable constructor into a `self` case for the classic ς self variable, and a `param` case for the parameter. While it might seem that allowing only one parameter is a limitation, that is actually not the case. The parameter can be a vector accumulating several values, so that the one variable denotes an arbitrary number of parameters; passing the empty object corresponds to passing no parameter.

The additional parameter is realized by introducing a new datatype for variables and changing the variable case of the `sterm` datatype accordingly. Moreover, the `Call` constructor for the method invocation was amended to include a parameter. Compare the datatype shown below to the datatype for ς_{ASC} in the definition below. All omitted constructors were kept unchanged.

Definition Syntax of $\varsigma_{Asc<:+}$

```
datatype Variable = Self nat | Param nat

datatype sterm =
    Var Variable
```

```
    ...
  | Call sterm Label sterm
    ...
```

An advantage of introducing the parameter as a new kind of variable is that variables of either kind can be handled in a single step. For instance, we defined the substitution in Section 5.1 for a single self variable, so that $t[s/0]$ replaces the variable with the index 0 in the outmost binding level[5] in t with the term s. As the identifier 0 is no longer unique, but refers to two variables – one self variable and one parameter variable – one might expect the introduction of a new substitution function for parameters. However, this is not necessary. As the variables are always substituted in the same step – method invocation – and have indices following the same rules, it is much easier to extend the substitution to handle both variables at the same time. This is realized by an extended substitution function, $t[u, v/0]$, that is structured exactly like the original substitution, but employs a new function for variables. The definition of the variable case is shown below, along with a stub of the substitution function. For the remaining cases, see the analogous definition in Section 5.1.

```
primrec
  subst_Var ::    "[Variable, sterm, sterm, nat] ⇒ dB"
where
   "subst_Var (Self i) a b k =
                          (if k < i then Var (Self (i - 1))
                           else if i = k then a else Var (Self i))"
   |"subst_Var (Param i) a b k =
                          (if k < i then Var (Param (i - 1))
                           else if i = k then b else Var (Param i))"

primrec
  subst        :: "[sterm, sterm, sterm, nat] ⇒ sterm"  ("_[_,_/_]")
and
  subst_option :: "[nat, sterm, sterm, sterm option] ⇒ sterm option"
where
  "(Var i)[s,t/k]       = subst_Var i s t k"
  |"(Call a l b)[s,t/k] = (Call (a[s,t/k]) l (b[s,t/k]))"
...
```

The value for the parameter variable is included in the same substitution step. We adapt `lift`, the operation used for adjusting de Bruijn indices, in a similar fashion; we leave the details of the adaption as an exercise to the reader[6]. Figure 7.3 shows the complete reduction relation for this extended

[5] Using de Bruijn indices, the index of a given variable increases with each traversed binder.

[6] See the Appendix for the solution.

calculus. Note how the BETA case of the new reduction relation uses the two-variable substitution.

$$\textsc{Beta}\ \frac{l \in \mathbf{dom}\ \ f}{\texttt{Call}\ (\texttt{Obj}\ f\ T)\ l\ a \rightarrow_{Asc} the(f\ l)[(\texttt{Obj}\ f\ T), a/0]}$$

$$\textsc{Upd}\ \frac{l \in \mathbf{dom}\ \ f}{\texttt{Upd}\ (\texttt{Obj}\ f\ T)\ l\ a \rightarrow_{Asc} \texttt{Obj}\ (f\ (l \mapsto a))\ T} \qquad \textsc{SelL}\ \frac{s \rightarrow_{Asc} t}{\texttt{Call}\ s\ l\ a \rightarrow_{Asc} \texttt{Call}\ t\ l\ a}$$

$$\textsc{SelR}\ \frac{a \rightarrow_{Asc} b}{\texttt{Call}\ t\ l\ a \rightarrow_{Asc} \texttt{Call}\ t\ l\ b} \qquad \textsc{UpdL}\ \frac{s \rightarrow_{Asc} t}{\texttt{Upd}\ s\ l\ u \rightarrow_{Asc} \texttt{Upd}\ t\ l\ u}$$

$$\textsc{UpdR}\ \frac{s \rightarrow_{Asc} t}{\texttt{Upd}\ u\ l\ s \rightarrow_{Asc} \texttt{Upd}\ u\ l\ t} \qquad \textsc{Obj}\ \frac{s \rightarrow_{Asc} t \qquad l \in \mathbf{dom}\ \ f}{\texttt{Obj}\ (f(l \mapsto s))\ T \rightarrow_{Asc} \texttt{Obj}\ (f(l \mapsto t))\ T}$$

$$\textsc{Asp}\ \frac{s \rightarrow_{Asc} t}{l\langle s \rangle \rightarrow_{Asc} l\langle t \rangle}$$

Figure 7.3: The inductive definition of the reduction relation for the $\varsigma_{ASC<:+}$ calculus with parameters.

7.3 Formalizing Depth Subtyping

With the basic calculus now featuring parameters, it might be expected that the conforming subtyping concept as introduced in Section 7.1 should provide a sound result. However, aspects and the ς update feature mean that this expected solution is not applicable without modification.

Example: Naïve Subtypes

To illustrate the problem encountered with subtyping and method updates, we present this motivating example. The problem encountered is that an update operation that *should* be invalid cannot be discovered by the type system, as subtypes and subsumption can allow the inference of a supertype. Even with the added method parameters, the calculus is not type safe when introducing pure contra-variance.

The reason is that the naïve implementation of subtypes, i.e. allowing return types to be more special and parameter types to be more general,

as shown in Figure 7.4, would lead to a type error due to the nature of the ς calculus, which allows the updating of methods. We omit the rules for the reflexive and transitive closure, they would be identical to the rules Sub-Trans and Sub-Refl in Figure 7.6. Keep in mind that because of subsumption an object that is a member of a type is also a member of all of that type's supertypes. That means that any operation valid on a member of the supertype has to be valid on all members of subtypes. This naturally includes updates and – in our case – aspects.

Sub-Obj

$$\frac{\forall l \in dom(B) \qquad \texttt{return}(A_l) <: \texttt{return}(B_l) \qquad \texttt{Param}(B_l) <: \texttt{Param}(A_l)}{A <: B}$$

Figure 7.4: A naïve subtyping relation.

The following example uses the notation of the ς calculus, which can be hard to understand for readers unfamiliar with the notation. Methods are denoted as $\varsigma(x, y)$, where x is the $self$ variable and y the added parameter. Fields omit the $\varsigma(x, y)$ notation, i.e. are denoted just with their values. In the example, problems caused by a naïve implementation of subtypes are showcased by using a well-typed term that nonetheless results in a stuck term – contradicting the progress property of a sound type system. The key issue is that any typed term can also be typed with any supertype of its original type. However, the method update operation is in conflict with that notion, as a method might be variant and thus have a different signature in a subtype, for instance accepting more parameters than the original signature. Thus, updating a method with a new body that is well-typed w.r.t. a supertype but not the particular subtype, can lead to a stuck program, despite being within the bounds of the type system.

To illustrate the problem in detail, assume the type `ColoredPoint` to be as shown underneath:

```
ColoredPoint ≡df [color: Color, Void;
                  get_color : Color, Void;
                  get_defaultcolor : Color, Void;]
```

Then assume the object `red_point`, which is of the type `ColoredPoint` to be defined as shown here:

```
red_point ≡df [color = red;
```

```
                get_color = ς(x,y) x.color;
                get_defaultcolor = ς(x,y) x.get_color;
                ] : ColoredPoint
```

And finally consider this update operation, which updates the method `get_color` with a new version that is type-wise and semantically incompatible with the object:

```
red_point.get_color ⇐ ς(x,y) y;
```

The original implementation of `red_point.get_color` does not expect a parameter other than $self$[7]; it just returns the object's `color` field. This does not hold true for the new version, which expects a parameter for use as return value. Quite clearly, the new method body does not match the signature for `get_color` in the type `ColoredPoint`, as that is `Color, Void` – it returns an object of type `Color` and expects no parameter. Thus, the resulting object of the update would look like this.

```
result ≡df [color = red;
            get_color = ς(x,y) y;
            get_defaultcolor = ς(x,y) x.get_color;
            ] : Point
```

With this result, any call to `result.get_defaultcolor` would become stuck, as it calls the updated `result.get_color` without the required parameter. Based on this, one would expect that the type system rejects the update operation; it certainly has to do it to be sound.

However, we can construct a supertype of `ColoredPoint`, called `Point` in this example, so that we can type the faulty update operation. Here is a possible realization of such a type `Point`:

```
Point        ≡df [get_color : Color, Color;
                  get_defaultcolor : Color, Void;]
```

Following the naïve subtype relation, it can be seen that `ColoredPoint` is indeed a subtype of `Point`. The method `get_color` is replaced with a more

[7]Strictly speaking, it does require a parameter, but it can be of any arbitrary type, including the empty object. We thus omit the parameter in cases where the empty object can be assumed to be the parameter.

general parameter type – `Void`, the empty and thus most general type. According to the co-variant relation realized in the naïve subtype relation, this is allowed. The other method, `get_defaultcolor`, has an identical signature in both types and can be ignored in this regard. Now, we can observe the issue: The new method body from the update operation matches the signature of `get_color` in the type `Point` – it returns a `Color` and expects a `Color`. Hence, the update is legal on an object of the type `Point`; our object `red_point` is of the type `ColoredPoint`, a subtype of `Point`. According to the principle of *subsumption*, this means that `red_point` can be treated as an object of the type `Point`. Moreover, it is possible to infer the type `Point` for the faulty update operation, i.e.

$$\vdash \texttt{(red\_point.get\_color} \Leftarrow \varsigma\texttt{(x,y) y) : Point}$$

In short, we have constructed an operation that, while perfectly valid for a true instance of `Point`, causes an error on our example term, which is of a subtype of `Point`, as shown in Figure 7.5.

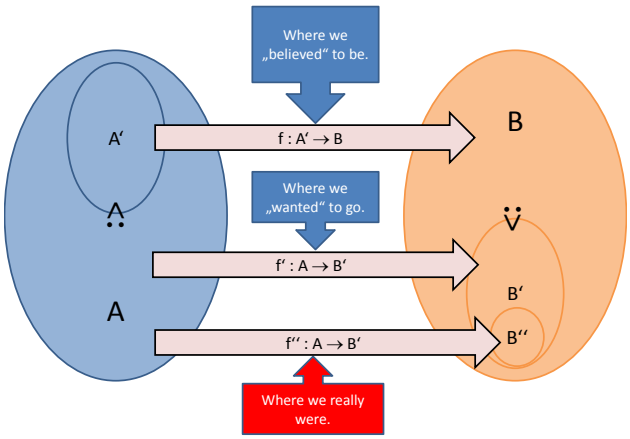

Figure 7.5: Variance issue encountered when using a naïve subtype relation.

Summarizing, we have shown how adding contra-variant – and thus generally assumed to be valid – subtyping harms type safety in our type system. It was possible to create a typed term that overwrites a method of an object with a method that expects a more special parameter type. Clearly, we have to introduce the means to correctly reject such instances from the type system.

Depth Subtyping with Variance Annotations

Considering the example, it becomes clear that the type of the new method has to be more special and more general at the same time, i.e. $A <: B$ and $B <: A$. By anti-symmetry of $<:$, this would mean that $A = B$.

One way around the problem presented in the example above would have been to remove the ability to update methods. After all, method updating is a feature not commonly found in mainstream object-oriented languages. However, it is a major feature of Aspect Orientation, thus we decided not to remove the update operation from the calculus.

Another way would have been to require invariance on methods, effectively removing variance and thus variance issues from the calculus. While there is precedence for such a step, it is deeply unsatisfactory by any practical standards – reducing the host calculi's flexibility is not expected of a concept striving to add flexibility.

Instead we used the approach of introducing a variance annotation for achieving type safety and maintain the expressivity needed. The use of a variance annotation is a tried concept for allowing method updating and variant subtyping at the same time [Pierce, 2002; Abadi and Cardelli, 1998]. It allows us to implement a clean subtype relation, which requires only minimal changes in the structure of the term typing rules. As such, we consider it a very natural and lightweight solution for achieving static type safety with non-trivial subtypes and method updating. A more direct approach, like establishing the most special type for each term in a typing judgment would have been possible as well. However, such typing rules would risk to be undecidable and are thus beyond a static type system. They would also have required major changes to the handling of types, which led us to the decision of using a variance annotation.

A number of changes to the type system were required to type method parameters, instead of the original limitation to return values. To support this, we changed the definition of the `type` datatype to the definition shown below:

```
datatype type =  Object "Label ↦ (type × type × bool)"
```

Instead of the original `type` datatype, which mapped method labels to types, the new construct maps method labels to `type` tuples. The idea behind the tuples is that the first entry performs the role of the original type, while the second entry acts as the type for the parameter. Finally, the third entry is a boolean flag, acting as variance annotation.

Figure 7.6 shows the new subtyping relation. Notice the changes to the precursors shown in Figure 7.4 and in Figure Figure 6.4.

$$\textsc{Sub-Obj}\quad \frac{\begin{array}{c}\forall l \in dom(B) \qquad \mathtt{return}(A_l) <: \mathtt{return}(B_l) \\ \neg\mathtt{variance}(A_l) \rightarrow (\mathtt{return}(A_l) = \mathtt{return}(B_l) \wedge \mathtt{Param}(A_l) = \mathtt{Param}(B_l)) \\ \mathtt{Param}(B_l) <: \mathtt{Param}(A_l) \qquad \mathtt{variance}(A_l) \rightarrow \mathtt{variance}(B_l)\end{array}}{A <: B}$$

$$\textsc{Sub-Trans}\quad \frac{A <: B \qquad B <: C}{A <: C} \qquad\qquad \textsc{Sub-Refl}\quad \frac{}{A <: A}$$

Figure 7.6: The co-inductive definition of the subtype relation for depth subtyping with a variance flag.

Our variance annotation differs from the annotations proposed by Abadi and Cardelli [Abadi and Cardelli, 1998], as the original ς calculus does not use explicit parameters. Hence it does not need annotations for parameters, but instead has to use a three state annotation to encode the different roles an object member can play (method, parameter, field, ...). We only use a boolean annotation variance for each member of a type, which either – if true – allows the conforming redefinition of a method's type, or – if false – does not. By this simple mechanism, the annotation enforces that there can be no inferred type for which a error-causing update operation would be valid. For updates to be safe, the flag has to guarantee that the new method is conforming to the original method without knowing the "real", most special type of the object to be updated. For fields, i.e. methods with the empty type as argument type, the flag poses no limitation whatsoever; the empty type is most general, so any update of fields is not hindered by the flag. The flag thus defines the exact signature against which to check methods in case of an update operation, adding the ability to safely change method signatures in subtypes without removing the unique update feature. It should be noted that the variance annotation itself is not invariant; a subtype can mark a method as invariant – and thus updatable – even when a supertype marked it as variant. However, the reverse is not true. The relation was formally shown to have all properties of a partial order: Reflexivity and transitivity by definition, anti-symmetry was proven. Just as with the original relation for width subtyping, we showed that the empty type is most general, being the supertype of all types.

A notable change is the environment construct, which now has to accommodate both parameter types as well as the original return value types. To achieve this, we change the definition to match that of the `type` datatype. The result is a construct that acts much like the one introduced for the basic type system, i.e. a stack that holds the type information for each index,

$$\textsc{VarSelf}\ \frac{x < |E| \qquad self(E,x) = B \qquad B <: A}{E, L \vdash \texttt{Var}\ (\texttt{Self}\ x) : A} \qquad \textsc{VarParam}\ \frac{x < |E| \qquad param(E,x) = B \qquad B <: A}{E, L \vdash \texttt{Var}\ (\texttt{Param}\ x) : A}$$

$$\textsc{Obj}\ \frac{\begin{array}{c}\mathbf{dom}\ b = \mathbf{dom}\ B \\ \forall i \in \mathbf{dom}\ B \qquad E\langle 0 : B, \texttt{Param}(B_i)\rangle, L \vdash b_i : \texttt{return}(B_i) \qquad B <: A\end{array}}{E, L \vdash \texttt{Obj}\ b\ B : A}$$

$$\textsc{Upd}\ \frac{\begin{array}{c}E, L \vdash a : B \qquad l \in \mathbf{dom}\ B \\ E\langle 0 : B, \texttt{Param}(B_l)\rangle, L \vdash n : \texttt{return}(B_l) \qquad B <: A \qquad \neg\texttt{variance}(B_l)\end{array}}{E, L \vdash \texttt{Upd}\ a\ l\ n : A}$$

$$\textsc{Call}\ \frac{E, L \vdash a : B \qquad l \in \mathbf{dom}\ B \qquad \texttt{return}(B_l) <: A \qquad E, L \vdash b : \texttt{Param}(B_l)}{E, L \vdash \texttt{Call}\ a\ l\ b : A}$$

$$\textsc{Lab}\ \frac{E, L \vdash a : B \qquad i < |L| \qquad L_i = B \qquad B <: A}{E, L \vdash i\langle a\rangle : A}$$

Figure 7.7: The inductive definition of the typing relation for depth subtyping.

following the de Bruijn indices of the variable. Any binder traversed in a typing judgment introduces an environment altering operation, $e\langle 0 : A, B\rangle$, where 0 means that the new entry is for the index 0 and that the self variable has the type A and the parameter variable the type B. The functions `Param` and `return` are used to access the parameter and – respectively – return type information from a type tuple.

Introducing the variance annotation and the altered constructs into the type system yields the typing relation shown in Figure 7.7. Note how the Upd uses the variance flag to reject update operations on variant methods, removing the variance issue from the base calculus. The relation is naturally closely related to the relation for width subtyping presented in Section 6.5. The Call rule was extended to require the parameter to be well typed. Note that methods not using the parameter can be typed by assuming the empty type for the parameter. The new relation establishes subsumption, i.e. the "is a" relation expected of subtypes.

Lemma 7.3.1 (Subsumption in $\varsigma_{Asc<:+}$)
$A <: B \wedge\ e, L\ \vdash\ t\ :\ A\ \Longrightarrow e, L\ \vdash\ t\ :\ B$

7.4 Properties of the Calculus with Depth Subtyping

While the introduction of parameters and depth subtyping extended the expressivity of the calculus significantly, it was a also a step introducing many changes to the inner workings. The extended calculus is still confluent, which was shown in a proof sharing the same basic outline with the one presented in Section 5.3. The handling of parameters required some alterations of the proof, but the property could be shown.

Aspect Compositionality, the other core theorem for the calculus without considering types, was also unaffected by the extension of the calculus, although the proofs needed some adjustment as well.

The soundness of the type system was of far greater interest, especially considering that a faulty subtyping relation would not yield a sound system. For instance, the naïve subtypes presented in Section 7.3 lead to an unsound system. Nonetheless, we were able to show the type soundness of $\varsigma_{Asc<+}$, which involved a number of new proofs regarding the handling of parameters and subtyping of variables. Most of the theorems ended up being significantly stronger than earlier incarnations, due to the inclusion of subtypes.

Considering the possible variance issues of Aspect Orientation, the most important result for $\varsigma_{Asc<:+}$ is that even the soundness of weaving, i.e. the type soundness for the assembled calculus remained valid. This means that the aspects benefit from the added variance, but the system is strong enough to prevent any ill-typed replacement of methods by aspects. We will explore this discovery in Chapter 9 [Kammüller and Sudhof, 2009b].

Also, even with parameters and depth subtypes, the type system is still strong enough to guarantee compositionality [Kammüller and Sudhof, 2009b], as it establishes the required $justoneFV$ property.

Thus, we were able to prove all properties of the simply typed calculus for the calculus with depth subtyping. Exceptions like the uniqueness of the typing relation come as no surprise, as the nature of subtypes removes such notions from the calculus. Note that terms still have a most special type established by the typing annotations at objects. We can thus state that the variance annotation, together with the modular typing judgment for aspects, solve the variance issue for Aspect Orientation in a clean fashion.

Summarizing this chapter, we introduced the concept of depth subtyping and variance in type systems. We showed problems with variance issues in real-world languages, including the leading aspect-oriented language, AspectJ. We further motivated the need for a non-trivial handling of variance by showing how a naïve realization of depth subtyping results in an unsound

type system. Building upon these observations, we introduced a sound notion of depth subtyping, employing variance annotations to remove the unsound cases from our calculus. Moreover, we introduced explicit parameters and parameter types to allow the realistic handling of variance. The resulting type system combined variance and type soundness, allowing us to expand the capabilities of the calculus significantly, while also proving that a sound solution for combining aspects and variance exists. Finally, we were able to re-establish all important properties that were shown for the significantly more limited calculi in previous chapters.

CHAPTER 8

Alternative Formalization: Locally Nameless Variables

This chapter marks a detour from the development of the ς_{ASC} calculus and introduces an alternative mechanization. We recommend to skip this chapter and to continue reading with Chapter 9: Connection to Reality.

As a contribution to the field of language meta-theory, this chapter presents a direct comparison of the locally nameless [Charguéraud, 2009] and de Bruijn [Bruijn, 1972] variable encoding approaches. Towards that end, we formalized the same calculus using the locally nameless representation (once more, in Isabelle/HOL). We only highlight the differences between the semantics and their effects on proofs in this section and do not re-introduce the calculus itself. The first part, showing how the operational semantics were formalized using the locally nameless representation, is presented in Section 8.1. The second part of this case study shows the adaption of the type system in Section 8.2.

The locally nameless representation poses a compromise between the de Bruijn representation and using names.The concept of locally nameless variables was previously introduced in Section 4.4. In the de Bruijn representation, all variables are represented by numeric indices, signifying their position in the structure of the term. By contrast, the locally nameless approach introduces named free variables for all instances that would call for index manipulation in the plain de Bruijn formalization [Henrio et al.]. Thus, it is possible to avoid the arithmetic proofs required for dealing with variable indices in systems using the pure de Bruijn representation.

To navigate from one variable type to another, there are two main variable-related operations on terms: *Opening* replaces bound variables with terms, combining the actions for the introduction of free variables and substitution. *Closing* on the other hand replaces a given free variable with a bound variable, using zero as the outmost binder; with each traversed counter the index is increased by one.

Based on these definitions, there are two predicates on terms, expressing basic well-formedness conditions required for working with terms. A term is *locally closed* if it has no bound variables pointing to a binder outside it, i.e. that there are no bound variables without a binder. There is a weaker condition required for methods in the ς calculus: *body*, which signifies that a term is a valid method body, i.e. expects to be surrounded by an object. The following section will introduce these operations and their applications.

8.1 Syntax And Semantics

The datatype representing the syntax is almost identical to the one formulated for the original version using de Bruijn indices in Definition 7.2, employing the same constructors for objects, terms, update and call operations as well as aspect labels. However, the cases for variables are notably changed to accommodate locally nameless variables. This development is joint work with our undergraduate student Bianca Lutz [Lutz, 2010]. Just like the de Bruijn version of the syntax, the constructor for bound variables – `Bvar` – can either represent a *self* or a *parameter* variable. By contrast, only one constructor is used for the named free variables. This is because the free variables are identified by their names, not their position in the term's structure. Discriminating between the self and parameter variables is required to avoid cases where two variables have the exact same position in the term's structure. Only numeric indices are affected by that, as names are required to be unique – fresh –, removing the possibility of such clashes.

```
datatype bVariable = Self nat | Param nat


types fVariable = string

datatype sterm =
    Fvar string
  | Bvar bVariable
  ...
```

Based on this datatype, we can introduce the operations mentioned above. The shorthand $t^{[s]}$ is used for $\{0 \longrightarrow [s]\}t$. Opening combines two key actions on terms, the first being to replace bound variables with free variables,

expressed $t^{[Fvar\ s]}$, and the normal substitution of variables. The realization in Isabelle/HOL takes the form of a primitive recursive function:

```
primrec
  sopen        :: "[nat, sterm, sterm, sterm] ⇒ sterm"
                              ("{_ ⟶ [_,_]} _")
and
  sopen_option :: "[nat, sterm, sterm, sterm option] ⇒ sterm option"
where
    "{k ⟶ [s,p]}(Bvar b)  = (case b of
                  (Self i)  ⟶ (if (k = i) then s else (Bvar b))
                  |(Param i) ⟶ (if (k = i) then p else (Bvar b)))"
  |"{k ⟶ [s,p]}(Fvar x)     = Fvar x"
  |"{k ⟶ [s,p]}(Call t l a) = Call ({k ⟶ [s,p]}t) l
                  ({k ⟶ [s,p]}a)"
  |"{k ⟶ [s,p]}(Upd t l u)  = Upd ({k ⟶ [s,p]}t) l
                  ({(k + 1) ⟶ [s,p]}u)"
  |"{k ⟶ [s,p]}(Obj f T)    =
                  (Obj (λ l. sopen_option (k + 1) s p (f l)) T)"
  |"{k ⟶ [s,p]}(l⟨ t⟩) = (l⟨{k ⟶ [s,p]}t⟩)"
  |"sopen_option k s p None     = None"
  |"sopen_option k s p (Some t) = Some ({k ⟶ [s,p]} t)"
```

The reverse operation for introducing free variables, closing, takes a similar shape.

```
primrec
  sclose        :: "[nat, fVariable, fVariable, sterm]        ⇒
                                    sterm" ("{_ ⟵ [_,_]} _")
and
  sclose_option :: "[nat, fVariable, fVariable, sterm option] ⇒
                                    sterm option"
where
   "{k ⟵ [s,p]}(Bvar b)     = Bvar b"
  |"{k ⟵ [s,p]}(Fvar x)     =
                  (if x = s then (Bvar (Self k))
                     else (if x = p then (Bvar (Param k))
                       else (Fvar x)))"
  |"{k ⟵ [s,p]}(Call t l a) =
                  Call ({k ⟵ [s,p]}t) l ({k ⟵ [s,p]}a)"
  |"{k ⟵ [s,p]}(Upd t l u)  =
                  Upd ({k ⟵ [s,p]}t) l ({(k + 1) ⟵ [s,p]}u)"
  |"{k ⟵ [s,p]}(Obj f T)    =
                  let (f' = (λ l. sclose_option (k + 1) s p (f l)))
                  in (Obj f' T)"
  |"{k ⟵ [s,p]}(l⟨ t⟩)       = l⟨({k ⟵ [s,p]}t)⟩"
  |"sclose_option k s p None     = None"
```

```
|"sclose_option k s p (Some t) = Some ({k ⟵ [s,p]}t)"
```

For convenience, we use the shorthand $\sigma[x]t$ for $\{0 \longleftarrow [x]\}t$. Closing introduces bound variables, i.e. variables identified by their position in the term structure, instead of the free variables matched by the name used on invocation. A pronounced advantage is the ability to express terms in a manner closely approximating their named equivalent on paper. Consider the trivial term/method $\varsigma(x).x$, which yields an object's self. The de Bruijn equivalent would be *var* 0, which is very different. Using the closing operation, we can write $\sigma[x].\ Fvar\ x$, which can be read as $\sigma[x].\ x$ – much closer to the original paper version. In fact, it would be feasible to use the exact $\varsigma(x)$ notation.

Our introduction of parameters complicates matters to a small degree. Much like the substitution on de Bruijn terms, we had to extend the opening and closing operations to handle parameters. Because of that, we use $\sigma[x,p]t$ for $\{0 \longleftarrow [x,p]\}t$, which closes the variable named x with the bound self variable of index 0 and the variable named p with the corresponding bound parameter variable. For opening, we use $t^{[x,p]}$ for $\{0 \longrightarrow [s,p]\}t$, which replaces the bound variables of index 0 with the terms s and p respectively.

Equipped with these two operations, we are almost ready to formulate the reduction relation in Locally Nameless style. However, there are a few issues left preventing us from formulating a meaningful relation. One problem that arises is that names, unlike structural indices, can conflict. For instance, *opening* is only a valid operation if it does not introduce free variables with a name already used in the term.

Cofinite Quantification

Otherwise, a *variable conflict* might occur: By instantiating a bound variable with an existing free variable of a surrounding context, the two become falsely identified. Such an identification introduces two issues. Primarily, the result will be faulty, as two variables that were not identical are identified. While this already provides ample reason for using fresh variable names, there is also another issue, namely that the reduction relation is no longer deterministic. With two variables using the same name, both are also captured by the respective other binder. Depending on which binder gets resolved first – completely arbitrary in a functional calculus such as ours – the result differs. Hence, whenever we have a rule that introduces a new variable, we need to ascertain that this name is fresh. In theory, the possibility to always assume fresh variable names is known as the Barendregt Variable Convention Barendregt [1984]. Defining a function FV to collect all names used in a term may seem like a possible solution to the problem. Technically, such a function is easily realized by traversing an entire term.

This way of formalizing can be described as the "exists-fresh" approach Aydemir et al. [2008]. For example, to express the previously sketched method reduction $\varsigma(x,y)t \to \varsigma(x,y)t'$ we need to presuppose that the variables x and y are fresh for t and t' otherwise we might end up with conflicting names.

The "exists-fresh" concept is straightforward and might seem like an ideal solution. However, it is not practicable in proofs, when rules are involved that reason about a part of a term behind a binder where the change of context caused by the use of *open* and *close* raises proof obligations for differing sets of free variables. In recent work by Aydemir et al. [2007], a very sophisticated technique called cofinite quantification was re-discovered that makes proofs involving such rules much clearer. The basic idea is to abstract from concrete sets of free variables $FV(t)$ and instead to consider some arbitrary finite set L, i.e. assuming a "cofinite set" of variable names. Since L is arbitrary, it can be chosen to be any kind of free variable set. To motivate the use of this technique in more detail, we consider a semantics rule whose formal introduction will occur later in this section. In the semantics of our object theory the abstraction binder is represented by the encapsulation in objects. The only other case where a binder occurs is in the second argument of an update. The abstraction encoded by close is used in the following informal update rule.

$$t \to_\varsigma t' \Rightarrow o.l := \varsigma(x,y)t \to_\varsigma o.l := \varsigma(x,y)t'$$

This rule is an example for the evaluation of a method body *under a binder*. In a locally nameless representation it is denoted as follows.

$$\frac{t^{[\mathtt{x},\ \mathtt{y}]} \to_\varsigma t'' \qquad t' = \varsigma[\mathtt{x},\ \mathtt{y}]t''' \qquad x{\neq}y \notin \mathtt{FV}\ t \qquad \mathtt{lc}\ o}{o.l := t \to_\varsigma o.l := t'}\ \text{UPDATE-LN}$$

We omit for simplicity the labeling `Fvar` in front of `x` and `y`. Note, that the bound variables are now opened in the hypothesis of the rule – the unintuitive structure using t'' as intermediary term is necessary because we do not know whether the reduction removes variables. The rule above is informal and does not make the origin of the two variables `x` and `y` precise. Informally, we can read the rules as *for at least one variable* `x` (or `y`, respectively) or as *for any variable* `x` (or `y`, respectively). Consequently, there are two interpretations of the above rule concerning the quantification of the `x` and `y`: *existential* or *universal* as follows.

$$\frac{t^{[\mathtt{x},\ \mathtt{y}]} \to_\varsigma t'' \qquad t' = \varsigma[\mathtt{x},\ \mathtt{y}]t'' \qquad x{\neq}y \notin \mathtt{FV}\ t \qquad \mathtt{lc}\ o}{o.l := t \to_\varsigma o.l := t'}\ \exists\text{-UPDATE-LN}$$

$$\frac{\forall xy.\ x{\neq}y \wedge x,y \notin \mathtt{FV} \ \wedge\ t \longrightarrow t' = \varsigma[\mathtt{x},\ \mathtt{y}]t'' \wedge \quad t^{[\mathtt{x},\ \mathtt{y}]} \to_\varsigma t'' \qquad \mathtt{lc}\ o}{o.l := t \to_\varsigma o.l := t'}\ \forall\text{-UPDATE-LN}$$

Unfortunately, neither of these rules is sufficient for reasoning. Since this rule, in either form, is intended to be part of the inductive definition of the

semantics, it can be read in two directions following induction or inversion, leading to corresponding introduction or elimination statements. However, in either, one direction of the resulting rules is too weak to be generally useful for reasoning.

A solution to this problem is the choice of stating the rule in a *cofinite* way as follows.

COFINITE-UPDATE-LN

$$\frac{\begin{array}{c} finite\ L \\ \forall x, y \neq y \wedge x, y \notin L \longrightarrow (\exists t''.0 \rightarrow_\varsigma [Fvar\ x, Fvar\ y]t = t'' \wedge t' = 0 \leftarrow [x, y]t'') \\ lc\ o \qquad body\ t \qquad t^{[\mathtt{x},\ \mathtt{y}]} \rightarrow_\varsigma t'' \end{array}}{o.l := t \rightarrow_\varsigma o.l := t'}$$

Here, as already intuitively described above, the concrete set `FV` t is abstracted into an (existential) finite set L. Reasoning about the freshness of names through the complement of a finite set of "used" names has already been recognized as an important concept by others, e.g. [Pitts, 2003], but the idea to directly integrate it into inductive definitions and thereby having a cofinite "induction" is new [Aydemir et al., 2007].

After having started out in a naïve way using locally nameless representation *without* using cofinite induction, we encountered several dead-ends when performing advanced proofs. The conclusion here is that the exists-fresh approach does not yield sufficiently strong rules for productive reasoning.

Thus, we overcome this limitation by employing a cofinite quantification to strengthen the rules. What initially may seem like an under-specification is actually sufficient for meta-theoretical reasoning: It expresses the same concept on a significantly more abstract level. Even more importantly, by under-specifying the set in the premises, we can use any finite set to instantiate the rules in proofs. The quantification is used when applying the rule by instantiating the variables and F with useful sets, for instance $F = FVt \cup \{a\}$, thus linking them to the properties required and derivable. On the technical side, in proofs, this instantiation is either done by instantiating an induction rule with well-chosen sets or by using the lemma shown below.

Lemma 8.1.1 (There are Fresh Names)
$finite\ L \implies \exists\ s\ p.s \notin L \wedge p \notin L \wedge s \neq p$

This lemma states that there are always names that are not in any given finite set. This lemma is used in a forward fashion to show the existence of fresh names under a strong assumption. Note that the variable L is usually not identified with the set raised by the induction, but by a derived set. For instance $L = F \cup \{a, b\}$, i.e. to yield two fresh names that are neither in the original finite F set and moreover different from a and b.

In beta reduction rules in Figure 8.1, the application of cofinite quantification can be identified by the premises $finite\ F$ and $x \notin F$.

Another obstacle to consider is the fact that locally nameless variables can express terms without a direct equivalent in the classical calculus. A bound variable pointing to a non-existing binder, for instance, is not the same as a free variable without a binder. Such binder-less variables are referred to as "dangling". As the indices of bound variables are constant throughout reduction in the locally nameless representation, it is possible that an initially "dangling" variable will be captured during the reduction, leading to faulty results. For this reason, it is necessary to restrict the reduction relation to terms with equivalents in the classical ς calculus. The predicate *lc* – locally closed – expresses this notion. Its use can be observed in the rules Beta, Upd, Beta, SelL, SelR and UpdR. It is defined as follows: The inductive predicate `lc` formalizes local closure:

$$\textsc{lc\_Fvar}\ \frac{}{lc(Fvar\ x)} \qquad \textsc{lc\_Call}\ \frac{lc\ t \qquad lc\ a}{lc\ (Call\ t\ l\ a)} \qquad \textsc{lc\_Lab}\ \frac{lc\ t}{lc\ n\langle t\rangle}$$

$$\textsc{lc\_Upd}\ \frac{lc\ t \qquad finite\ L \qquad \forall s\ p.s \notin L\ \wedge\ p \notin\ L\ \wedge\ s \neq p\ \longrightarrow lc(u^{[Fvars,Fvarp]})}{lc\ (Upd\ t\ l\ u)}$$

$$\textsc{lc\_Obj}\ \frac{finite\ L \qquad \forall l \in dom\ f.\forall s\ p.s \notin L\ \wedge\ p \notin\ L\ \wedge\ s \neq p\ \longrightarrow lc\ ((f_l)^{[Fvars,Fvarp]})}{lc(Obj\ f\ T)}$$

Closely related to this notion of *closed terms* is the predicate *body* – it expresses that the term is a method body, having at most a single unbound variable of each flavour - self or parameter. Restricting the term to one variable means that, if *body t* is true, then $t^{[\texttt{Fvar}x,\texttt{Fvar}y]}$ is guaranteed to be locally closed. For instance, a locally closed object has only methods for which the *body* predicate is true.

body
$body \equiv_{df} \exists L.finite\ L \wedge (\forall s\ p.s \notin L \wedge p \notin L \wedge s \neq p \longrightarrow lc(t^{[Fvar\ s,Fvar\ p]}))$

Note that no such limitation was required for plain de Bruijn terms, despite the similar mismatch between terms expressible and terms valid on paper. In either case, regardless of the variable representation, it is possible to express all practically relevant terms.

The complete reduction relation is shown in Figure 8.1. Especially consider the cases where a binder has to be traversed, i.e. the case for the

$$\textsc{Beta}\ \frac{l \in \mathbf{dom}\ f \qquad \mathtt{lc}(\mathtt{Obj}\ f\ T) \qquad \mathtt{lc}a}{\mathtt{Call}\ (\mathtt{Obj}\ f\ T)\ l\ a \to_{Asc} the(f\ l)^{[(\mathtt{Obj}\ f\ T),a]}}$$

$$\textsc{Upd}\ \frac{l \in \mathbf{dom}\ f \qquad \mathtt{lc}(\mathtt{Obj}\ f\ T) \qquad \mathtt{body}\ a}{\mathtt{Upd}\ (\mathtt{Obj}\ f\ T)\ l\ a \to_{Asc} \mathtt{Obj}\ (f\ (l \mapsto a))\ T} \qquad \textsc{SelL}\ \frac{s \to_{Asc} t \qquad \mathtt{lc}u}{\mathtt{Call}\ s\ l\ u \to_{Asc} \mathtt{Call}\ t\ l\ u}$$

$$\textsc{SelR}\ \frac{s \to_{Asc} t \qquad \mathtt{lc}u}{\mathtt{Call}\ u\ l\ s \to_{Asc} \mathtt{Call}\ u\ l\ t} \qquad \textsc{UpdL}\ \frac{s \to_{Asc} t \qquad \mathtt{body}u}{\mathtt{Upd}\ s\ l\ u \to_{Asc} \mathtt{Upd}\ t\ l\ u}$$

$$\textsc{UpdR}\ \frac{y \notin F \qquad x \neq y \qquad \mathtt{Finite}F \qquad x \notin F \qquad \exists t''.t^{[\mathtt{Fvar}x,\mathtt{Fvar}y]} \to_{Asc} t'' \wedge t' = \sigma[x,y]t'' \qquad \mathtt{lc}u}{\mathtt{Upd}\ u\ l\ t \to_{Asc} \mathtt{Upd}\ u\ l\ t'}$$

$$\textsc{Obj}\ \frac{x \neq y \qquad l \in \mathbf{dom}\ f \qquad \mathtt{Finite}F \qquad x \notin F \qquad y \notin F \qquad \exists t''.t^{[\mathtt{Fvar}x,\mathtt{Fvar}y]} \to_{Asc} t'' \wedge t' = \sigma[x,y]t'' \qquad \mathtt{lc}(\mathtt{Obj}\ f\ T)}{\mathtt{Obj}\ (f(l \mapsto t))\ T \to_{Asc} \mathtt{Obj}\ (f(l \mapsto t'))\ T}$$

$$\textsc{Asp}\ \frac{s \to_{Asc} t}{l\langle s \rangle \to_{Asc} l\langle t \rangle}$$

Figure 8.1: The inductive definition of the reduction relation using locally nameless variables.

reduction in the parameter of the update operation , UpdR, and the one for reduction inside an object Obj. In these cases, terms have to be opened with fresh variables to keep them locally closed. The rules never explicitly link the set F to the free variables of a term, that connection is only established when using the rules in a proof. The only statement about F is that it is finite and thus that there is an infinite number of fresh variable names. This cofinite quantification keeps the rules flexible and useful for proofs.

In the rules using cofinite quantification, the introduction of two variables at once, self and parameter, leads to an explosion in the number of premises in these cases. The statement $\exists t''.t^{[\mathtt{Fvar}x,\mathtt{Fvar}y]} \to_{Asc} t'' \wedge t' = \sigma[x,y]t''$ found in the premises of those rules – UpdR and Obj – is required, as reduction only applies to locally closed terms. These two rules describe the reduction within methods, thus the methods have to be closed before any reduction is possible.

Comparing the Beta rule to its sibling in Figure 7.3, it is clear how opening is used to fill the role of the substitution operation.

While it can be argued that this set F can be confusing for readers not familiar with proof tactical considerations, it is also fairly easy to see that the statement expresses a notion of variable freshness. On the other hand, writing $\sigma[x].x$ for a method is much more straightforward than \$0.

A result of using lc in the premises is that only terms which are locally closed can reduce, a required limitation, i.e.

Lemma 8.1.2 (Only closed terms reduce)
$s \rightarrow_{Asc} t \longrightarrow lc\ s \wedge lc\ t$

This is also know as the *regularity* of the reduction relation [Charguéraud, 2009].

With the formalization established, it was possible to prove the confluence of the reduction in a proof similar to 5.3. In fact, the proof benefited significantly from the use of locally nameless variables and ended up being shorter than the original by about 600 lines of proof script or about a quarter of its total size.

8.2 Types

For further investigation, we adapted the type system for $\varsigma_{Asc<:+}$ to locally nameless variables as well. Our major hope in this step was to come up with a type system that handles variables in a more intuitive way, while also being able to simplify and generalize the many index permutation lemmas required for the original type soundness proofs.

Not surprisingly, the typing relation for locally nameless variables requires a very different approach to environments than the one that was used in the typing relation for de Bruijn index based terms. The first fundamental change is that we base the typing of locally nameless variables entirely on opened terms, i.e. there is no way for bound variables to be in the type system. This implies that the environments have to map names to types, instead of the old stack based environments, which related the type information solely by the composition of the stack. Now, the environments for locally nameless variables have to relate names to types.

This is further complicated by the requirements of being able to formulate strong rules with cofinite quantification, so that it cannot be allowed to overwrite an entry in an environment. Such restrictions are hard to impose with a normal map, so that instead the environments are realized as a new datatype. This type is not just based on simple dictionaries, but is stateful insofar as that it has an error state. The datatype is defined as follows:

```
datatype α environment =
      Env fVariable ↦ α
    | Malformed
```

Based on this datatype, we define the function to add new elements to the environment, the equivalent for the shift operation in the de Bruijn version of the type system. As we track free variables, which are not split into parameter and self variables, we do not require tuples of types to be added at once, but only single types[1]. The add function is realized as a primitive recursive function:

```
primrec
    add :: α environment ⇒ fVariable ⇒ α ⇒ α environment (_(|_:_|))
where
    (Env e)(|x:A|) = (if (x ∉ dom  e) then (Env e(x ↦ A))
                     else Malformed
    Malformed(|x:A|) = Malformed
```

The idea here is that inserting a duplicate element leads to the error condition from which the environment cannot be recovered. I.e. $(Env\ e)(\!|x : A|\!)$ is valid, iff x is not in the domain of e; $(Env\ e)(\!|x : A|\!)(\!|x : B|\!)$ leads to the error state.

The statement that an environment is not in an error state is expressed by the predicate *ok e*, which states that e is finite and not malformed. Note that adding an element does not impede finiteness, i.e. as long as the environment started out as finite – empty – the add operation cannot cause it to become infinite.

Well-formedness of Environments
$ok\ e \longrightarrow finite\ e\ \wedge\ e \neq Malformed$

An important property of the add operation is that it is commutative:

Lemma 8.2.1 (Add is Commutative)
$e(\!|x : A|\!)(\!|y : B|\!) = e(\!|y : B|\!)(\!|x : A|\!)$

A notable benefit of this environment style is that it would be possible to handle type variables for polymorphic terms without changes to the typing environments.

Using this environment construct, we can define the typing relation, reusing the subtyping relation from Figure 7.6.

[1] Being polymorphic, the environment construct is able to handle tuples.

$$\textsc{Var}\quad \frac{ok E \qquad x \in \mathbf{dom}\ E \qquad E_x = B \qquad B <: A}{E, L \vdash \mathtt{Var}\ (\mathtt{Fvar}\ x) : A}$$

$$\textsc{Obj}\quad \frac{\begin{array}{c} finite F \\ ok\ E \qquad \mathbf{dom}\ b = \mathbf{dom}\ B \qquad \forall i \in \mathbf{dom}\ B \qquad s \notin F \qquad p \notin F \qquad s \neq p \\ E(\!|s : B|\!), (\!|\mathtt{Param}(B_i)|\!), L \vdash b_i^{[\mathtt{Fvar}\ s, \mathtt{Fvar}\ p]} : \mathtt{return}(B_i) \qquad B <: A \end{array}}{E, L \vdash \mathtt{Obj}\ b\ B : A}$$

$$\textsc{Upd}\quad \frac{\begin{array}{c} finite\ F \qquad E, L \vdash a : B \qquad l \in \mathbf{dom}\ B \qquad s \notin F \qquad p \notin F \\ s \neq p \qquad E(\!|s : B|\!), (\!|\mathtt{Param}(B_i)|\!), L \vdash n^{[\mathtt{Fvar}\ s, \mathtt{Fvar}\ p]} : \mathtt{return}(B_l) \\ B <: A \qquad \neg \mathtt{variance}(B_l) \end{array}}{E, L \vdash \mathtt{Upd}\ a\ l\ n : A}$$

$$\textsc{Call}\quad \frac{E, L \vdash a : B \qquad l \in \mathbf{dom}\ B \qquad \mathtt{return}(B_l) <: A \qquad E, L \vdash b : \mathtt{Param}(B_l)}{E, L \vdash \mathtt{Call}\ a\ l\ b\ :\ A}$$

$$\textsc{Lab}\quad \frac{E, L \vdash a : B \qquad i < |L| \qquad L_i = B \qquad B <: A}{E, L \vdash i\langle a \rangle : A}$$

Figure 8.2: The inductive definition of the typing relation for depth subtyping, using locally nameless variables.

Properties

The first important property of the locally nameless typing relation is its regularity, using the regularity term as per Charguéraud [2009]:

Lemma 8.2.2 (Typing is regular)
$E, L \vdash\ t\ :\ A \Longrightarrow ok\ e\ \wedge\ lc\ t$

This property has no equivalent in the de Bruijn based relation, but is absolutely essential for the locally nameless version. Another such infrastructural theorem is the renaming lemma, which asserts that choosing different – fresh – names does not alter typing. It is the locally nameless equivalent of the lemma required in a named approach to deal with α conversion, but thankfully easier to prove, using cofinite quantification.

The ς base of the calculus added an interesting requirement to the environment premises in the typing judgments. The published examples for locally nameless variables use System F as example, where no self referencing is involved. In System F, it is sufficient to add the *ok E* premise to the

variable case(s). With our calculus, however, it is necessary to add $ok\ E$ to the premises of the Object case, as the regularity of the system would not be derivable otherwise.

Lemma 8.2.3 (Renaming Lemma)

$$\begin{array}{ll} & E(\!|s : A|\!)(\!|p : B|\!), L \vdash \{bv \to [Fvar\ s, Fvar\ p]\}t\ :\ T \\ \wedge & s, p \notin FV\ t\ \wedge x, y \notin FV\ t\ \wedge p \neq s\ \wedge\ x \neq y \\ \Longrightarrow & E(\!|x : A|\!)(\!|y : B|\!), L \vdash \{bv \to [Fvar\ x, Fvar\ y]\}t\ :\ T \end{array}$$

The lemma above states that, if a term is typed after opening it with two names s, p, it is also well typed using any other fresh names. Note that $s \neq p$ follows from the term being well-typed in an Environment $E(\!|s : A|\!)(\!|p : B|\!)$. For proving type safety, the lemma had to be strengthened to allow arbitrary well typed terms instead of just new free variables.

Lemma 8.2.4 (Opening Lemma)

$$\begin{array}{ll} & E(\!|s : A|\!)(\!|p : B|\!), L \vdash \{bv \to [Fvar\ s, Fvar\ p]\}t\ :\ T \\ \wedge & s, p \notin FV\ t\ \cup FV\ x \cup FV\ y \\ \wedge & E, L\ \vdash\ x\ :\ A \wedge E, L\ \vdash\ y\ :\ B \\ \Longrightarrow & E, L \vdash \{bv \to [x, y]\}t\ :\ T \end{array}$$

Using these lemmas as infrastructure, it is possible to establish type soundness by following the general schema introduced in earlier chapters.

Aspects are constructed in a manner similar to the one used with de Bruijn indices. As mentioned before, the self variable 0 was used as entry point for the base term and is substituted in weaving. Using locally nameless variables, it would be possible to use a bound variable, maintaining the de Bruijn system. However, typing does not recognize bound variables, so that any typing notion would require opening the terms first. Thus we consider it to be much cleaner to introduce a name for the base call, $base$[2], so that the typing predicate for aspects now reads:

Well-Formed Aspects

$$wf\_asp\ T\ \langle pc, adv\rangle \equiv_{df} \forall l \in\ dom\ T.\emptyset(\!|base : T_l|\!), L \vdash\ advA\ :\ T_l$$

Using a "magic" variable name does not require that variable $base$ to be fresh in the base term. This becomes clearer when considering the weaving function: Weaving can now be expressed using the substitution function, following the same algorithm as in Chapter 5, so that if a pointcut matches, the advice adv is woven into a term t by:

```
[base ⟶ t]adv
```

[2]Terms like *continue* or *proceed* would have been viable alternatives, but misleading about the nature of the calculus.

The substitution replaces the variable *base* in the advice. Since possible other occurrences of the same name in the term t are irrelevant for typing and semantics, this means there are no restrictions on variable names in base terms required other than the requirements already imposed. The advantage of the names being independent is that there is no requirement to share environment data other than the label between the aspect and the base typing – just as with the de Bruijn indices. The proofs for the compositionality of weaving were not completely performed for the locally nameless version. We expect the proofs to be comparable to the de Bruijn formalization.

8.3 Discussion of the Locally Nameless Formalization

Overall, the result of using the locally nameless representations are mixed. The removal of the lifting operation leads to much more readable proofs, while the cofinite quantification eased some, but not all proofs about terms and relations. For instance, the proof of the Church-Rosser property is shorter by about a third, while the basic proofs and definitions are longer by a similar margin. The reduction relation's definition is less readable in some regards, and easier to read in others, but maintains a close link to the de Bruijn notation in all cases. In fact, it would be feasible to reduce a de Bruijn term with the locally nameless representation, using solely bound variables for the original numeric variables. Generally, the locally nameless semantics are more abstract than the semantics based on de Bruijn indices. The cofinite quantification only requires an arbitrary finite set to be available – it is up to an implementation to identify the abstract finite set with the free variables in the terms. Such an identification could even use renaming, implementing variable scopes and shadowing without contradicting the semantics.

A glaring difference between the formalizations is the requirement to add well-formedness conditions to both, the reduction and typing relations. While the de Bruijn formalization as developed in the preceding chapters of this path does not require additional predicates in the premises of the rules, we had to add the requirement to be locally closed, *lc* to the reduction relation and the requirement for the environment to be valid – *ok* – to typing. Both of these predicates have their origin in the variable handling. The requirement to be locally closed states that no bound variables without a binder exist in a term, i.e. no "dangling" variables [Charguéraud, 2009]. While dangling variables are possible in the de Bruijn representation, as remarked in Section 4.4, these do not generally pose a problem for the reduction, as the lifting operation maintains their dangling nature. In the locally nameless representation however, such a dangling variable might be

captured by a binder during the reduction, leading to a faulty result. By contrast, the well-formedness requirement for typing environments is rooted in the notion of freshness for the names of free variables. The *ok* predicate guarantees that no two entries for the same name exist in the typing environment. This guarantees in turn that no name for a variable was used twice during opening. Thus, the type system enforces variable freshness due to the well-formedness condition. The de Bruijn representation dispensed with names completely, which also removed any concerns about the freshness of names – hence, the premise is not required for de Bruijn variables.

We consider the locally nameless representation a valuable tool that makes several proofs far more readable. The juggling with indices is one of the prime cause of errors in de Bruijn based formalizations, a problem effectively solved by the locally nameless representation. Additional factors, such as code extraction, can be seen as points in favor of de Bruijn indices at the moment.

Concluding this small deviation from the main topic of the thesis, we have shown that the locally nameless representation is a viable one for both, the ς calculus and derived approaches and for language theory in Isabelle/HOL.

CHAPTER 9

Connection to Reality

After the very technical presentation of the various ς_{Asc} calculi in the preceding chapters, we wish to show the practical connection to the reality of programming in this chapter. A formal calculus such as the one introduced in this thesis is naturally on a far higher level of abstraction than a language intended solely for practical programming. However, it has the rigorous approach most practical languages are missing and thus can be used as a model to show where scenarios occur that impede the safety of a concept. Towards the end of connecting our formal system to the real domain of programming, we use a two-tiered approach: Code generation from the formal method is used to make the ς_{Asc} calculus itself usable as a language, bridging the gap from the formal system to reality. The gap in the other direction, representing real problems in relation to the calculus, is handled by re-creating critical situations in – for example – AspectJ in the calculus. This shows that the calculus is able to express the situation and that the solution presented in this thesis prevents the scenario from causing errors. Figure 9 visualizes this concept of connecting the calculus to the domain of mainstream programming languages.

Using these results, the final section of the chapter constructs a categorization of aspects. This categorization expresses what aspects can be represented in the calculi presented earlier, which are safe and compositional and why certain aspects fall outside the scope of this thesis.

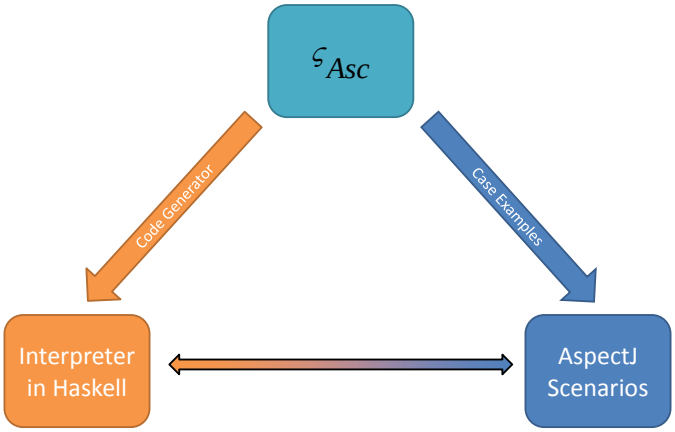

Figure 9.1: Connecting ς_{Asc} to Reality.

9.1 Code Generation – Type Checker and Interpreter

A well-established way of creating a bridge between a formalization and programming is to provide an interpreter for the designed calculus [Dantas and Walker, 2006]. There are two established approaches[1] for showing the correlation between an interpreter and a calculus.

The first such method is to program the interpreter by hand and then formally show the equivalence between the calculus and the implementation. This approach can be applied to languages not mechanized in an interactive theorem prover, by manually performing the necessary proofs. Nonetheless, even the manual translation into a programming language can benefit from using such an interactive theorem prover. A recently introduced tool, the Haskabelle importer [Haftmann, 2009], allows the import of programs written in the functional language Haskell into Isabelle/HOL. Using this tool, it is possible to formally prove the refinement relation between a program and a formalization.

The second such approach is to use a mechanization in an interactive theorem prover to extract executable code from it. This has the benefit that the code created is in fact identical to the formalization in the proof assistant, establishing a very strong link between code and formalization that has the added benefit of easier maintainability – changes in the formalization are

[1] We ignore the "it's an interpreter because we say that it is" approach, which arguably also is established.

automatically echoed in the extracted code. Both, Coq and Isabelle/HOL, support the feature of extracting code from formalizations. Coq formalizations, as long as they limit themselves to the set of constructive features, are even programs right way and directly executable [Letouzey, 2008] – a benefit of using a constructive logic as base formalism. Isabelle/HOL was recently extended to extract code in a proven toolchain, not just from recursive functions, but also from inductive definitions [Berghofer et al., 2009].

Linking code to a formalization is becoming more and more important in software engineering [Kammüller, 2006], regardless of the chosen style of such a connection. Today, it is even feasible to program entire systems inside a theorem prover, delivering a formal system as the actual product. The advantage is that entire programs can be proven to be correct w.r.t a set of correctness properties. Either style of adding formal proofs of correctness is often seen as one of the next big steps towards zero defect software. Examples of programs completely verified include a L4 Kernel [Klein et al., 2009], termination proof checkers [Thiemann and Sternagel, 2009] and compilers [Leroy, 2009]. Using monad based input and output, even user interfaces can be constructed in the formalization, although the more established approach is to wrap the proven code with a user interface.

For ς_{Asc}, we choose the approach of extracting code from the formalization [Hakobyan, 2010], using the state of the art code generation from inductive functions [Berghofer et al., 2009] [2]. The new Isabelle/HOL code generator does not merely generate code, but builds upon a formal equational system to guarantee that the code refines the formalization. Possible target languages include SML and Haskell, but even imperative languages are targeted for future versions.

Prior to generating code from an inductive definition in Isabelle/HOL, it is necessary to provide a version of said definition that is suitable for the generator. "Suitable" in this context means that the generator is able to analyze the rules of the definition and to find an interpretation of the inductive definition that is consistent, i.e. one that can deduce which variables are input and which are output.

For instance, an interpreter would be a 1:1 relation – one entry term is evaluated to one reduction result. Some logical operators make this analysis complicated, for instance premises about domains of maps, quantifiers ($\forall$, $\exists$) or premises that are not sufficiently constructive. To bridge the gap between "suitable" and the actual formalization, it is sometimes required to offer alternate definitions of the inductive definitions, which can – for instance – use specialties like an enumerable type to solve quantification and domain

[2] We thank Lukas Bulwahn of the Isabelle group at Technische Universität München for providing us with a developer version of an advanced code generator, as well as for his valuable help with using the tool.

questions or to use functions that solve premises constructively. Consider this rule as an example for such a rewritten judgment, which replaces the normal BETA rule from the reduction relation:

```
lemma beta_new: "⟦Predicate.eq (Some v) (f l)⟧
  ⟹ "Call (Obj f T) l p →Asc v[(Obj f T),p/0]"
```

The tool will present the user with proof obligations to show that the definitions are, in fact, in a refinement relation. Due to the early state of these tools, the learning curve for establishing which additional information is required, or which premise needs rewriting is very steep. Nonetheless, the function of extracting code is worth the effort and yields remarkable insights, not to mention a very tangible connection to reality.

For our calculus, we were able to extract code from the beta reduction, using redefinitions from the non-trivial reduction rules. The underlying approach taken was that we assumed the method labels to be from a finite, enumerable set. Using this premise, it is possible to resolve some quantors and reasoning about elements being keys in a finite map – used to model objects.

To yield an usable program, the actual interpreter is combined with a pretty-printer. This step was taken as the code resulting from the generation only implements the one step reduction relation. However, a usable interpreter requires a multi-step interpretation. In a more user-oriented approach, it would be entirely feasible to construct a user interface inside Isabelle with monads used for input and output.

For the type system, similar steps were required. However, the final assembly ended up being more complicated due to the use of all-quantification in the OBJ rule. The required workaround relies on the same foundation as the reduction: An enumerable domain makes it possible to iterate not just over all methods, but over all possible methods. In the context of the type checker, an older change to the calculus was re-used: The Object annotations – originally intended to introduce type uniqueness – are technically sufficient to construct the type. This step allows us to not just extract a type checker – a four parameter function with a boolean result – from the code, but even a type inference tool – a tool that infers the type of a given term, i.e. a 3 parameter function with a type as result.

Summarizing, we can report that the extraction of code from the formal model was both possible and successful in connecting the calculus to the reality of programming. The work and experience required was non-trivial, which leads us to state that the extraction of code from inductive definitions is not yet advanced enough for use in software development, but will be in the

near future. Especially the code generation from functions and equational lemmas was easy to use and yielded reproducible results. On the other hand, the generation from inductive definitions required non-trivial adjustments.

9.2 Case Examples

The extraction of interpreter and typechecker presented in the preceding section is an important part of our claim that our calculus has real-world applications. While a complete translation from one language into our calculus is not feasible, we are able to translate scenarios that are known to be of interest into our calculus. That ability to express these scenarios is a strong case for the applicability of the approach. We argue that the problems are indeed captured by the calculus, as the problems arise on the type level. Even in our functional calculus, which is naturally unable to capture reference problems, the typing problems are identical. By solving the type issues in the corresponding situations, we are able to extend the solutions to real-world languages. The notation used throughout this section is based on the original ς calculus [Abadi and Cardelli, 1998]. In this notation, fields are shown without the ς binder, as they do not use any variables. Furthermore, for methods, we write $varsigma(x, y).m$ for a method with the $self$ parameter x and the argument y; m is an arbitrary method body. $Void$ is used to denote the empty, most general type, which is used to indicate that a given parameter is not used; the empty object is assumed as parameter in these instances.

Observers

As a first example, we show how an aspect counting the invocations of methods can be realized. As a first example, we present a simple aspect that counts the invocations of a method. The following AspectJ aspect realizes such a functionality:

```
public aspect Counter {
static int counter = 0;
  Object around() : call(* someMethod(..)) {
    counter++;
    return proceed();
  }
}
```

Represented in ς_{Asc}, the aspect takes the form shown below: We abstract from de Bruijn indices in all of the examples, using the normal ς notation

instead. For advice, we use the notation $Adv(x).$, where the x represents the base variable.

```
asp ≡df {[...]; Adv (base).[counter = zero;
                      counter_inc = ς(x,y) x.counter ⇐ x.counter.succ;
                      count       = ς(x,y) x.counter_inc.proceed;
                      proceed     = ς(x,y) base;
                    ].count }
```

This aspect `asp` works by replacing a labeled method with a new object wrapping a counter and a call to a method of that new object. The aspect's method `count` then invokes the originally intercepted base term, continuing the original control flow. The aspect also indicates one of the properties of our calculus: It is entirely functional and has no mutable state. As a result, such an aspect would natively neither be able to maintain a global count, nor able to "catch" invocations on copies of the original object[3] as those would have their own counts. However, that can be easily overcome by using an artificial count during the reduction of the term, i.e. the evaluation is able to maintain a global count, albeit external to the calculus. This corresponds to logging into a file or onto the console, which is also realized by adding "print" pseudo-methods.

Dynamic Pointcuts – cflow

Another example for a limitation on the ς_{Asc} calculus is the concept of dynamic pointcuts. Such dynamic pointcuts can be used to link the weaving of an aspect to conditions like the method from which another method call originated or to boolean guards.

In ς_{Asc}, it is conceivable that "`if`" pointcuts – pointcuts including a guard predicate – can be realized by boolean conditions in the advice itself, for in a formal calculus, there is no logical difference between a guard evaluated during weaving and one in the advice itself.

As a more complex example, we have to consider `cflow` and related pointcuts – pointcuts that are dependent on the call history of a given joinpoint. Consider the pointcut used in the aspect below, which states that the aspect should only act if the method `setColor` was called from the method `move`.

```
public aspect Cflow {
  Object around() :
```

[3]To work with a method resembling function, the aspect would also have to be rewritten to increment the counter on the return value, i.e. a post-increment, instead of the pre-increment realized above.

```
      call(* setColor(..)) && withincode(call (* move(..))) {
   ...
  }
}
```

A static analysis can easily resolve the pointcut if the method `setColor` is either always or never called from the context of the method `move`. In the former case, the aspect is always woven, in the latter never. This concept of resolving dynamic pointcuts statically was developed even further by Avgustinov et al. [Avgustinov et al., 2007]. This approach of statically resolving dynamic pointcuts can easily be applied to ς_{Asc}: A primitive recursive function could resolve the dynamic pointcuts and places labels accordingly.

Another viable way of expressing some control-flow related pointcuts in ς_{Asc} is the use of exhaustive labeling. Consider the following ς_{Asc} example. It contains three labels: Label 1 marks the method `move`, label 2 marks the method `setColor`. Neither of these labels corresponds to a situation comparable to the `withinCode` pointcut above. However, the label 3 marks the invocation of `setColor` from within `move`, perfectly matching the desired pointcut.

```
ColoredPoint ≡df [move     = ς(x,y) 1⟨...2⟨x.setColor(red)⟩...⟩
           setColor = ς(x,y) 3⟨x.move;⟩
           ...
          ] : Point
```

Naturally we cannot claim that all dynamic pointcuts are expressible in our calculus. However, we can express and resolve a large number of dynamic pointcuts. Note that we only considered the possible encodings of dynamic pointcuts in our calculus, but not static details, especially typing. That is because dynamic pointcuts can be ignored for questions for type safety. Typing is a static concept and thus has to handle dynamic pointcuts statically. This also means that as far as typing is concerned, the dynamic part of any pointcut has to be considered to be true. This also means, that any safety or security property guaranteed by the type system holds true for dynamic pointcuts as well.

Summarizing, we can express a number of dynamic pointcuts, either by using guard expressions or static analysis, so that many – albeit not all – scenarios can be examined.

Types Prevent Mistakes

Types were introduced in Chapter 6 as a tool to prevent errors. We revisit the example from the introduction to show that the type system indeed catches errors that would cause runtime problems otherwise.

```
public class ColoredPoint
{
color = Color.green;

  public Color getColor()
  {
    return this.color;
  }

  ...
}

public aspect asp
{
  Object around() : call(* *.getColor(..))
  {
    return "oops";
  }
}
```

Figure 9.2: A not type safe program in AspectJ.

The example in Figure 9.2 shows a non-typesafe aspect in AspectJ. A method returning an instance of the type "`Color`" is replaced by a method returning a `string` instead. `Color` and `string` are in no subtype relation with each other, any assignment of a `Color` to a `String` variable or vice-versa constitutes an error, resulting in a `ClassCastException`. The program will thus terminate with an `Exception` in `Test.test`. This breaches modular reasoning, as the base class `Test` is sound on its own.

For the translation into the calculus, we employ a few simplifications to maintain readability. The first such adaption is that we assume a type "`Color`" to exist; `green` is assumed to be a member of this type. We furthermore assume that strings have the type `String` expressing the same concept as in Java and other programming languages. Finally, we abstract from de Bruijn indices by using named variables.

Translated into ς_{Asc}, the same situation looks like this:

```
point ≡df [color     = green;
           get_color = ς(x,y) 1⟨x.color;⟩
           ...
          ] : ColoredPoint
```

We labeled the method `get_color` with the label 1 without adding any further explicit labels. In the general case, labeling would be complete. The type `ColoredPoint` is structured accordingly:

```
ColoredPoint ≡df [color     = Color, void;
                  get_color = Color, void;
                  ...
                 ]
```

And the aspect:

```
asp ≡df {[1]; "oops"}
```

We can see that weaving indeed leads to the same result as it does in AspectJ:

```
point⇓asp = [color     = green;
             get_color = ς(x,y) 1⟨"oops"⟩
             ...
            ] : ColoredPoint
```

Furthermore, this term is clearly no longer well-typed – the string is not typable as `Color,void`. Consequentially, the question arises, whether typing would catch this error. The answer is the label environment L, which enforces aspect and base compatibility. For the method `get_color` to have the type `Color` – required to type the whole object as `ColoredPoint` –, the label Environment has to contain `Color` or a subtype thereof as entry for the label 1. This is expressed by the LAB rule in the type system:

$$\text{LAB}\quad \frac{E, L \vdash a : L_i \qquad i < |L| \qquad L_i <: B}{E, L \vdash i\langle a \rangle : B}$$

Conversely, the aspect is well typed, if it is type preserving w.r.t. the same label interface. The typing rule for aspects states that an aspect has to be typed according to the label types for all labels captured by its pointcut:

```
wf_asp L_T ⟨pc, adv⟩ ≡df
        ∀ l∈dom  L_T. []⟨0:L_T !l⟩, L_T ⊢ adv: L_T!l
```

In the given example, this evaluates to

$$[]\langle \texttt{0:Color,void}\rangle\ \texttt{L}_\texttt{T} \vdash \texttt{"oops"} : \texttt{Color}$$

According to our premise that strings are not a subtype of `Color`, the aspect is shown to be incompatible with the label environment and thus ill-typed. Hence, the type system catches the error that would have caused the runtime environment to crash.

Solving Contravariance

The motivation for introducing variance into the calculus was already extensively discussed in Chapter 7, however, at the time the effect of the variance annotation was solely shown by proving type soundness. Now we revisit the example and show why an aspect can complicate matters in the presence of the redefinition of method types in subtypes.

For the fleshed out example, we use the following scenario: A class `Point` defines a field `origin`, storing the origin of the current coordinate system. This field is accessed via the public `getOrigin` method. In a subclass of `Point`, `ColoredPoint`, the method is re-defined to return a different kind of value: An instance of the very same subclass. The addition of an aspect overwriting the method `getOrigin` in `Point` now yields a problem: Is the aspect legal, despite potentially cutting across several methods with different types? To make the aspect as directed as possible, the example also uses a `target` pointcut to explicitly state the expected type and omits the pointcut subtyping operator `+`. The example code omits some method definitions, like the two-argument constructor for the class `Point`. To enhance readability, we limit the example to the methods causing the error.

```
public class Point {
  int x;
  int y;
  Point origin;

  public Point getOrigin() {
    return origin;
  }
  ...
}
public class ColoredPoint extends Point {
  ColoredPoint coloredOrigin;

  public ColoredPoint getOrigin() {
    return coloredOrigin;
```

```
  }
  ...
}

public aspect asp {
  Object around() : call(Point Point.getOrigin(..)) && target(Point) {
    return new Point(0,0);
  }
}
```

The question raised by this example can also be expressed visually, as shown in Figure 9.3.

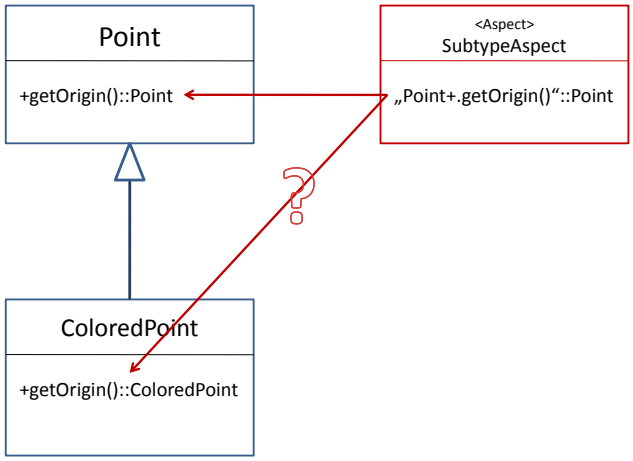

Figure 9.3: Should the aspect apply to subtypes? Can it?

The answer in AspectJ is clear: When the method `ColoredPoint.getOrigin()` is called, the aspect returns the incorrect type. The result is a `ClassCastException`, which does only mention the call to `ColoredPoint.getOrigin()`. This is deeply unsatisfactory, as the trace omits any indications of the real error: The presence of an unsound aspect.

For the translation of this scenario, we assume that a type “int” exists; it can be assumed to express the same concept as `int` in Java and is not of central importance for the example. Moreover, we assume `cart_origin` and `cart_green_origin` to be constants of the type `Point` and respectively `ColoredPoint`. Otherwise, the same preconditions as in the preceding example apply.

```
point ≡df [origin    = cart_origin;
           x         = 0;
```

```
 y         = 0;
 getOrigin = ς(x,y) 1⟨x.origin;⟩
 ...
] : Point
```

Types in $\varsigma_{Asc<:+}$ consist of a triple: A return type, a parameter type and a variance annotation. We use "`-`" for an invariant type and "`+`" for a variant one. I.e. a method returning an `int`, expecting a `string` that could be re-defined in a subtype would have the signature `x,y,+`. The type `Point` is thus structured as shown below, the fields are updateable and thus have the variance "`-`" – the method `getOrigin` is variant, as it is to be redefined.

```
Point ≡df [origin    = Point, void,-;
           x         = int,void,-;
           y         = int,void,-;
           getOrigin = Point, void,+;
           ...
          ]
```

And the `ColoredPoint` translates like this:

```
colored_point ≡df [origin        = cart_origin;
                   coloredOrigin = cart_green_origin;
                   x             = 0;
                   y             = 0;
                   color         = green;
                   getOrigin     = ς(x,y) 2⟨x.coloredOrigin;⟩
                   ...
                  ] : ColoredPoint
```

And the type:

```
ColoredPoint ≡df [origin        = Point, void,-;
                  coloredOrigin = Point, void,-;
                  x             = int,void,-;
                  y             = int,void,-;
                  color         = Color;
                  getOrigin     = ColoredPoint,void,+;
                  ...
                 ]
```

It is easy to see that `ColoredPoint` <: `Point` holds true: The types are identical for the features `x`, `y`, `origin`, satisfying the reflexivity of subtyping.

The features `color` and `coloredOrigin` only exist in `ColoredPoint` – as subtypes are always allowed to be larger – width subtyping. Finally, the method `getOrigin` was co-variantly re-defined, now using the more special `ColoredPoint` as return type. As the variance flag for the method is true, this redefinition is arguably well-typed. This statically easy situation becomes interesting when adding the aspect:

```
asp ≡df {[1,2]; [origin    = cart_origin;
                 x         = 0;
                 y         = 0;
                 getOrigin = ς(x,y) 1⟨x.origin;⟩
                 ...
                ] : Point
         }
```

We can infer that the label environment has to hold the type `Point` for the label 1 and `ColoredPoint` for the label 2. The entirely static advice in this aspect is typed as `Point` or any supertype thereof, regardless of the base term. The label 2 marks a term of the type `ColoredPoint`, so that the application of the aspect with the given label environment is not well typed as `Point` is not conforming to `ColoredPoint` – the type system catches this potential covariance situation without even "breaking a sweat". Note that it would be well-typed to use an aspect having the type `ColoredPoint` in the given label environment.

Updating with Aspects

The preceding examples showed how the calculus copes with static values overwriting methods and how the type system catches such instances. It should be noted that the preceding examples did not in fact use the variance annotation and thus did not make clear how variance annotations interact with aspects. As entry point, we use a slight variant of the preceding example: Instead of statically returning a new object, we now use an aspect that interacts with the base application.

We use the objects and types from the preceding example with only a slight change. The new aspect replaces the `Point.getOrigin` method with an implementation accessing the called object's state:

```
public class Point {
  Point origin;

  public Point getOrigin() {
    return new Point();
  }
```

```
  ...
}

public class ColoredPoint extends Point {
  ColoredPoint coloredOrigin;

  public ColoredPoint getOrigin() {
    return coloredOrigin;
  }
  ...
}

public aspect asp {
  Object around() : call(Point Point.getOrigin(..)) {
    Point point =  (Point)thisJoinPoint.getTarget();
    return point.origin;
  }
}
```

Note that the AspectJ aspect does not use typed, direct access to the called object; a reflection method with a cast is required[4].

In ς_{Asc} there are two subtly different ways of altering a given object, both of which behave very differently when it comes to typing. The first method, as used in the preceding two examples, is an aspect replacing a method with a static value. Such a replacement can even happen on fields that are not variant according to their type, i.e. any member of an object can be overwritten by such an aspect. This is safe for two reasons: The first and most important reason is that an aspect applied on the method level cannot access the object itself, nor method parameters passed upon invocation of the replaced method. At the same time, its type is strictly enforced by the label type, i.e. an application of such an aspect to a whole subtype hierarchy would not be well-typed. The aspect type has to be at least as special as the original method.

The second method is more powerful: An aspect can invoke a method update on the entire object, replacing object members with completely new implementations and without added restrictions. This type of aspect can be applied to members of a type, including subtype instances. Such flexibility is possible, because the aspect has to abide by the rules of the basic type system and hence to respect the variance annotations introduced for this very reason.

To flesh this situation out, we analyze the example above by considering several viable translations into ς_{Asc}. We assume the label 1 to label the whole

[4]Using `target` pointcuts, the type can be established, which does not solve potential variance issues.

object `point`, while the label 2 labels the method `getOrigin`. The labels 3 and 4 are used for `coloredPoint` accordingly. Ignoring the additional labels, the objects are unchanged from the previous example and thus omitted here. The first aspect realization, using the straightforward method replacement:

```
asp_replace ≡df {[1,3]; ς(x,y) x.origin; }
```

Laying out the aspect like this captures the functionality nicely, but is not well typed: The aspect well-formedness condition requires an aspect to be well typed in an empty environment, i.e. aspects are not allowed to access any information from outside their label's scope. This is not a simple limitation, but an essential trait that enables the calculus to simulate different weaving semantics. Our solution to this apparent impasse is quite simple: The scope has to be declared properly. Indeed, using the labels 2 and 4, we are able to express the functionality in a well-typed manner.

```
asp_update ≡df {[2,4]; Adv (x) x.getOrigin ⇐ ς(a,b) a.origin; }
```

This version is well-typed, as it accesses information in its scope: Namely the feature `origin` of the labeled object. This version respects the variance annotation introduced earlier, with the exact intent of allowing the type safe handling of aspects like the above example. Moreover, this style of expression corresponds to introductions, the ability of aspects to add new methods to objects, as found in AspectJ.

9.3 A Classification of Aspects

The various limitations and problems of aspects and the ways to express them in our calculus open another question: Does the calculus establish a classification of aspects insofar as that different kinds of aspects can be expressed in the various incarnations? Our tentative answer is yes, as we can link more "safe" behavior with smaller subsets of aspects. As such, each more "special" set of aspects is a subset of all more "general" sets.

Our classification begins with the most general aspects, i.e. the full power of AspectJ and other real-world languages. As this domain includes aspects that rely on reference semantics, mutable state and similar concepts, it is not possible to capture them entirely in a functional calculus like ς_{Asc}.

In this group of aspects, it can be observed that some aspects cannot be expressed in a functional fashion directly. Instead, they can be brought in line with the functional model by assuming that some functions have side-effects. This assumption and even realization of side-effects can mean that proofs about the calculus are not entirely applicable if the side-effects

influence the evaluation. Generally, assumed functionality such as output can be simulated without a significant loss of rigorousness.

The first group entirely expressible in ς_{Asc} is the group of functional aspects, i.e. the aspects that can be translated into a functional representation, without state or side effects. They can further be specialized into the group of compositional aspects, the class of aspects which provably exhibit the same behavior regardless of the applied pointcut and weaving semantics.

The well-typed aspects form the next smaller group, known to be compositional but also known not to cause any crashes or inconsistent program configurations. Safe aspects are even more restrictive – these aspects do not even alter the result of the program, but merely implement functions like the logging or tracing in programs. The semantics of the advised program are not affected by this and safe aspects are proven. The final, most restrictive set of aspects are harmless, where harmless is taken from the concept of harmless advice [Dantas and Walker, 2006]. Harmless advice maintains noninterference [Sabelfeld and Myers, 2003], a strong information flow security property.

As a summary, the different classes of aspects are shown in Figure 9.4, ranging from most general at the bottom to most restrictive at the top.

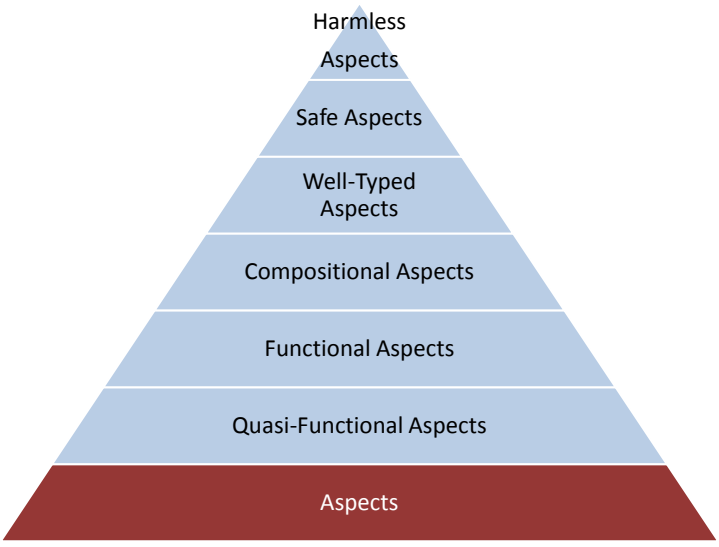

Figure 9.4: Aspects, from most general to most special.

9.4 Comparison to Application-Oriented Concepts

The work presented in this thesis is oriented towards establishing a formal calculus for the study of Aspect Orientation and to show how such a small calculus can be employed to analyze issues, predominantly typing issues, in established languages. It is further intended as a testbed for innovative concepts, offering a base known to be sound. This is the reason why other core calculi designed with comparable intentions were considered as examples for related work in Chapter 3. In the spirit of this more practically oriented chapter, we add a short collection of related work that uses similar concepts or has related goals from an application-oriented perspective.

Shmuel Katz et al. model aspects and base programs as state machines [Katz, 2006; Katz and Katz, 2009], enabling them to verify temporal logic formulae automatically with a model checker. This approach is then used to construct a statically decidable categorization of aspects that recognizes spectative aspects, regulative aspects and invasive aspects. Invasive aspects are further divided into weakly and strongly invasive aspects, depending on their impact on the base semantics. In either case, the categorization is correlated with a loss of modularity. Nonetheless, modular verification of even strongly invasive aspects is explicitly supported in recent work [Katz and Katz, 2009]. We consider the categorization of aspects to be related to our classification, as the general properties of the different classes move along the same lines. Moreover, the use of mechanized formal methods and the weaving semantics are comparable to our approach. Particularly interesting is the use of a model checker, which constitutes – along with interactive theorem provers – a very important approach for the computer based verification of models.

Steimann et al introduce joinpoint types [Steimann et al., 2009] as a new first-class citizen in an implicit invocation based language (IIIA). Their approach is inspired by the Java approach for exceptions, as it requires explicit declaration of joinpoints in the base program. This approach has the benefit of maintaining a stronger form of modularity than is generally expected in Aspect Orientation. We believe that this approach is closely related to the one presented in this thesis, as the typed explicit labels employed in ς_{Asc} are a comparable concept. Both concepts, IIA and ς_{Asc} support a finer granularity of joinpoints than just methods by allowing the explicit exposure of joinpoints in the control flow and polymorphic behavior in advice. Moreover, the inheritance/subtyping semantics are similar insofar as that joinpoints are not necessarily inherited. While the approach of Steimann et al is much more oriented towards practical programming, we believe that ς_{Asc} is a viable formalism for future research into IIIA.

Avgustinov et al. developed a static tool to resolve AspectJ pointcuts [Avgustinov et al., 2007]. Their approach is based on a translation of AspectJ pointcuts into queries for the prolog-derived database language datalog, which yield the places to be advised in a base program. We see a close correlation between this concept and ours, as the idea of introducing labels hinges on the assumed existence of an algorithm to automatically insert the labels at the required locations of the base application. We propose a concept inspired by this one in Section 9.2.

ObjectTeams [Herrmann, 2007] is a collaboration-based programming language with a strong focus on type soundness. Based on Java, it adds the new construct team to the language, which can be seen as a combination of package and class. Roles, members of the team, can be linked to types in the base application, which also allows them to alter the program semantics in an aspect-oriented fashion. Unlike AspectJ, ObjectTeams does not employ a pointcut language, but instead links roles to base types – and role instances to base instances – using fully qualified types and method names. This is similar to the label approach proposed in this work, as it requires the unequivocal identification of the joinpoints to use. ObjectTeams handles the variance situation shown in Section 9.2 by using an unmentionable type. We see some similarity between the polymorphic typing judgment used for aspects in ς_{Asc} and the unmentionable type in ObjectTeams, which is, in fact, a direct influence of the research behind this work into the ObjectTeams project.

Gudmundson and Kiczales propose the use of explicit [Gudmundson and Kiczales, 2001] joinpoint interfaces – interfaces detailing all joinpoints exposed in a given module – to restore modularity and allow the clean selection of appropriate joinpoints. We argue that this is semantically equivalent to explicit labels and our joinpoint typing interface.

Mezini and Kiczales offer a slightly altered concept [Kiczales and Mezini, 2005], by suggesting that such an interface could be composed after the program is completed, i.e. aspects upsetting the original structure were added. These aspect-aware interfaces would allow reasoning about typing issues and bears some resemblance to Avgustinov et al [Avgustinov et al., 2007]. From our perspective, either approach is compatible with ς_{Asc} and its notion of label interfaces. The major difference in the representation of the various approaches is in the assumptions about labels: Are they a) explicitly inserted or b) automatically generated or c) complete.

By establishing that code generation is able to extract usable code from the formal specification, we have established that the formal calculus represents an executable language. By further expressing key examples for problematic situations in the industry leading language AspectJ in our calculus, we have shown that its expressivity is sufficient to reason about such

situations. Finally, we have related our calculus to the reality of Aspect Orientation by offering possible ways to connect it to state-of-the-art approaches and identifying different classes of aspects.

CHAPTER 10

Conclusion

This chapter completes the thesis by summarizing and evaluating the results presented earlier. We begin with a short summary of the preceding chapters. We then formulate the contribution of this thesis, detailing the results. The remainder of this chapter is used for a more personal perspective, expressing lessons learned and giving thanks where due.

10.1 Summary

This work bridges two different domains of computer science, applying the domain of rigorous language development in interactive theorem provers to the practically oriented domain of Aspect Orientation. The bridge constructed in this work is a simple, aspect-oriented calculus that due to its simplicity is well-suited for the mechanization in a theorem prover. At the same time, the calculus allows the concise and natural representation of aspect-oriented languages.

The foundation of the work was built throughout Part I, where we motivated our research into the safety of Aspect Orientation and presented pre-existing aspect-oriented calculi. We compared these calculi regarding their expressivity, intent and features in a survey. By analyzing the survey, we came to the conclusion that no aspect-oriented calculus was ever mechanized in a theorem prover, a key requirement for our approach to rigorous language analysis. Moreover, we found that most calculi are either not

object-oriented or very closely linked to Java and AspectJ. Even of object-oriented aspect calculi, only one explicitly considered subtyping and variance issues.

We thus formulated our goal to present a mechanized calculus that is simple to use and natively handles objects as first class citizens. An additional requirement was not to be limited to representing a single language, but to remain as general as possible. Considering different alternatives, we picked the ς calculus as basis for our calculus. The ς calculus combines the requirements of being mature, object-oriented, extensible and not limited to a specific language, which made it an excellent fit.

Based on these decisions, we presented the construction of our calculus in Part II. The first incarnation of the calculus, shown in Chapter 5, is untyped. Key features of the untyped calculus are the natural and simple representation of aspects and weaving, using an approach of explicit joinpoints in the form of labeled subterms. Expanding on these constructs, we were able to show core properties, especially the compositionality of weaving and the confluence of the core calculus.

We then continued to establish safety and soundness for the calculus, especially as static properties. Types were the tool of our choice to establish these static properties. We presented a modular type system for aspects in Chapter 6 that unifies static typing for base terms with a polymorphic typing approach for aspects. Building on that notion of typing, we were able to prove the soundness of our type system – including aspects. In a further extension, we extended the type system to allow subtyping, maintaining strong type soundness while relaxing the restrictions imposed by types.

By considering the situations where real aspect-oriented programs behave contrary to the expectations of static typing, we concluded that variance issues are a major problem with aspects. To accommodate variance issues, we extended the calculus even further in Chapter 7. This extension allowed us to formalize depth subtyping, which is a known issue in real-world languages such as AspectJ. Using our modular notion of aspect typing, we were able to re-establish static type soundness by introducing variance annotations.

A connection to reality was established in Chapter 9, where we detailed the ability to extract executable code from our formal model using the code generation feature of Isabelle/HOL. To further strengthen the connection, we constructed equivalent formulations of real-world snippets in our calculus, explaining why our type system is able to detect various error situations. Based on these experiences, we presented a classification of aspects with regards to our notion of safety.

Beyond the development of the aspect-oriented calculus, we also mechanized a version of the calculus using the modern approach of locally nameless variables, presented in Chapter 8.

10.2 Contribution

In this thesis we have presented several achievements: A simple concept for modular aspects, a mechanized core calculus for the study of Aspect Orientation and a formal proof of type soundness for aspects in an environment with depth subtyping. These achievements are complemented by achievements in the field of language meta-theory, notably the comparative formalization using different binder concepts and code generation. Finally, we presented a classification of aspects based on our formalism.

We can thus present a number of primary findings, which are key results of the formalism without considering the technical contribution in the field of language theory.

- We developed the ς_{Asc} family of calculi, which constitute a simple, yet expressive set of core calculi for Aspect Orientation.
- Among the core features of these calculi is a simple concept for aspect typing, which realizes aspect and base typing on the basis of a shared label or joinpoint interface.
- We defined and proved a concept of Aspect Compositionality
- We focused out attention on subtyping, formalizing both width and depth subtyping.
- We proved the soundness of the aspect typing and showed that a type system using variance flags is suitable for solving the variance issues present in real-world languages, such as AspectJ.
- Based on our findings, we realized a classification of aspects.

We further contributed to the field of language theory. The mechanizations are to be submitted to the archive of formal proofs.

- Completely mechanized the calculi in Isabelle/HOL in the tradition of a conservative extension.
- Compared different variable representations, using the locally nameless and de Bruijn representations for variables.
- Our formalization served as testbed calculus for proven code generation in Isabelle/HOL.

10.3 Lessons Learned

The research behind this thesis began a little over three years ago, when a study in aspect re-usability [Sudhof, 2006] found severe problems with aspect modularity and aspect re-usability. The project ASCOT presented itself as the ideal venue to study and solve the issues encountered. Then, we sought to build a formal model to prove the safety of collaboration based languages, notably ObjectTeams [Herrmann, 2007]. After the initial surveying of the situation and the tools at hand, the first lesson learned was an old one: "Keep it simple." Simplicity is the most important tool in formal research and in a variation of Occam's razor it can be said: "The simplest formalism able to represent a given domain is also the best formalism to represent that domain."

Simplicity keeps models small and understandable, allowing extensions and thus flexibility. It also is something that does not come without a price: A "simple" model is not simpler to create, it is easier to use.

The second major lesson revolves around the use of interactive theorem provers for users with a pre-dominantly practical background. These tools are invaluable for rigorous research and ease the connection of formal models to real domains. However, their use is, unlike other automatic proof tools, notably Model Checkers, not well suited for trial and error. There is not always the possibility of automatically generating a counter example, nor can the inability to complete a proof be seen as equivalent to the invalidity of the lemma to be proven.

The use of theorem provers requires a steep learning curve, both in order to master the logics employed in those tools and to master the technology. Beyond that, time must be invested to learn the proof libraries shipped with the prover distributions. Even after all these steps, the instability of the provers can introduce problems: In general, provers are not backward compatible. Every new version of a prover requires significant changes to existing proofs. This made it especially difficult – even more difficult than for formal topics in general – to find students willing to work with theorem provers or to write their thesis about topics related to theorem proving.

Generally, we feel that simplicity in a formalization is a requirement for clean theories. This is even more important in a rigorous environment, like a theorem prover, as such tools add a considerable overhead on their own. We feel that even considering these drawbacks, the use of a theorem is a worthwhile addition to the process of formalizing a language. This is in part due to the added guarantees and in part due to the eased connection to executable programs.

Finally, one observation became more and more obvious while writing this thesis: Some area is always left untouched. While it would be possible to compile a long list of future work, it would also be dishonest to the spirit of this work to do so. This thesis tried to find a equilibrium, with a calculus being both small and expressive. However, there is at least one notable angle that warrants future research: The combination of the ASP_{fun} calculus for distributed objects [Henrio et al., 2007] and ς_{Asc}, leading to a theory of distributed aspect components. We hope to be able to revisit this project in future work.

10.4 Acknowledgments

This thesis and the underlying research is a result of the DFG project ASCOT (Grant Ja 379/18-1) and would not have been possible without it. Several people offered insights, research and ideas for this thesis and we want to express our thanks for those who contributed.

Florian Kammüller of the Technische Universität Berlin contributed advice, motivation and help on uncountable occasions. He deserves the most pronounced thanks for his support and guidance, which made this thesis a reality. Also, our thanks to Stefan Jähnichen, also of the Technische Universität Berlin, who made the project ASCOT possible. My compliments to the people who collaborated on various topics throughout the research of this thesis: Ludovic Henrio of the INRIA Sophia-Antipolis for the joint work on the ASP_{Fun} Calculus [Henrio et al., 2007]. Lucas Buhlman of the Technische Universität München tweaked the Isabelle code Generation to meet the needs of the calculus introduced in this thesis [Berghofer et al., 2009] and my graduate student Lilit Hakobyan, who initiated the work on code generation. My undergraduate student, Bianca Lutz, who worked on the Locally Nameless variable representation. Thanks to Pascal Fradet of the INRIA Rhône-Alpes, who asked interesting questions about aspect compositionality and to Michael Mendler, who provided valuable feedback for the present, published version of this thesis. Also, I wish to express my gratitude to the many people in the PES and SWT groups at Technische Universität Berlin, who contributed insights and ideas. Finally, my heartfelt thanks to those who proofread the many drafts of this work. Thank you, Florian, Margaretha, Samuel, Eva, Ana and Carolina.

10.5 Closing Remarks

In this final paragraph, we wish to bring attention to the original goal of this thesis: The rigorous proof that aspects can be modular and sound. We

presented a formal calculus that realizes a safe concept of Aspect Orientation with special attention to modularity and type soundness. However, we do not claim that this automatically guarantees the safety of all aspects. This thesis, and others, have proven the opposite. What we claim is that we can express a safe class of aspects in our calculus, forming a subset of aspects in general. We believe that a large number of practically relevant aspects is included in that category and that Aspect Orientation as a whole benefits from the rigorous approach to language development.

APPENDIX A

Complete Formalization Using de Bruijn Indices

This appendix presents the actual Isabelle/HOL theories that form the basis for this work, omitting the proof bodies. For the formalization using locally nameless variables and the complete proofs, we refer to our homepage [Kammüller and Sudhof, 2009a]. This appendix was automatically generated by Isabelle and presents the mechanization as it was validated by Isabelle/HOL, without any editing. The syntax can be confusing for users not familiar with the notations used in Isabelle/HOL, particularly the interaction of kernel and object logic. For a complete introduction into the syntax, we refer to the Isabelle tutorial by Nipkow et al [Nipkow et al., 2002]. As a simplified tour we repeat the short introduction from 2.3: The semantic brackets enclose premises for lemmas, using the meta implication as pseudo-conjunction. $[\![A; B]\!] \Longrightarrow P$ stands for $A \Longrightarrow (B \Longrightarrow C)$ and can be read as $A \wedge B \longrightarrow C$.

The appendix is structured as follows: Omitting essential properties of lists and other trivial preliminaries, we begin by presenting the finite map type used throughout the calculus. Having established that core building block, we continue with the ς calculus, especially syntax, semantics and basic properties. As a final finding for the core calculus, we then present the confluence proof. Types and Type Soundness are the topic of the next section, starting with type environments, subtyping, the type system and finally the theorems for subject reduction and progress. Aspects then are first introduced, defining their syntax as well as weaving; we close with the compositionality property and the various soundness proofs. Additional proofs and formalizations are available on our homepage [Kammüller and Sudhof, 2009a].

A.1 Finite Maps with Axclasses

theory *FiniteMaps* **imports** *Preliminaries* **begin**

Initial setup of the type as axiomatic type class.

axclass *fintype < type*

finite_set: "finite (UNIV)"

types *('a, 'b)fmap = "('a :: fintype) ~=> 'b"* (**infixl** *"-~>" 50*)

Induction on finite sets: Has to be valid for the empty set and the finite set to which one arbitrary addition was made.

theorem *fset_induct:*
"P {} ==> (!!x (F :: ('a :: fintype)set). x ∉ F ==> P F ==> P (insert x F)) ==> P F"
⟨*proof*⟩

Uniqueness of finite maps.

theorem *fmap_unique: "x = y ⟹ (f :: ('a, 'b)fmap) x = f y"*
⟨*proof*⟩

A finite map is either empty or there is another finite map that is identical if one entry is added to it.

theorem *fmap_case:*
"(F :: ('a -~> 'b)) = empty ∨ (∃ x y (F' :: ('a -~> 'b)). F = F'(x ↦ y))"
⟨*proof*⟩

Define the witness as a constant function so it may be used in the proof of the induction scheme below. Splits a fmap into domain and range.

constdefs
*set_fmap :: "'a -~> 'b ⇒ ('a * 'b)set"*
"set_fmap F == { (x, y). x : dom F & F x = Some y}"

Predicates can be lifted to sets.

constdefs
*pred_set_fmap :: "(('a -~> 'b) ⇒ bool) ⇒ (('a * 'b)set) ⇒ bool"*
"pred_set_fmap P == % S. P (% x. if x : fst ‘ S then (THE y. (? z. y = Some z & (x, z): S)) else None)"

Removing an entry.

constdefs
*fmap_minus_direct :: "[('a -~> 'b), ('a * 'b)] ⇒ ('a -~> 'b)"* (**infixl** *"--" 50*)

```
  "F -- x == (% z. if (fst x = z & ((F (fst x)) = Some (snd x))) then
None else (F z))"
```

An entry that got added is in the map.

lemma *insert_lem : "insert x A = B ⟹ x : B"*
⟨*proof*⟩

Translation lemma for members of set_fmap.

lemma *set_fmap_pair: "x: set_fmap F ⟹ (fst x : dom F & snd x = the (F (fst x)))"*
⟨*proof*⟩

Removing and adding the same entry is neutral.

lemma *set_fmap_inv1: "⟦ fst x: dom F; snd x = the (F (fst x)) ⟧ ⟹ (F -- x)(fst x ↦ snd x) = F"*
⟨*proof*⟩

Order of adding an entry and set_fmap can be reversed.

lemma *set_fmap_inv2: "fst x ~: dom F ⟹ insert x (set_fmap F) = set_fmap (F (fst x ↦ snd x))"*
⟨*proof*⟩

A predicate lifted to a set is logically identical.

lemma *rep_fmap_base: "P (F :: 'a -~> 'b) = (pred_set_fmap P)(set_fmap F)"*
⟨*proof*⟩

Usable version of the above.

lemma *rep_fmap: "(? (Fp :: ('a * 'b)set) (P' :: ('a * 'b)set ⇒ bool). P (F:: 'a -~> 'b) = P' Fp)"*
⟨*proof*⟩

A finite set is finite.

theorem *finite_fsets: "finite (F :: (('a :: fintype) set))"*
⟨*proof*⟩

The domain of a finite map is finite.

lemma *finite_dom_fmap: "finite ((dom (F :: 'a -~> 'b)) :: (('a :: fintype) set))"*
⟨*proof*⟩

Range of a finite map is finite.

lemma *finite_fmap_ran: "finite (ran (F :: ('a :: fintype) -~> 'b))"*
⟨*proof*⟩

The set representation of a finite map is finite.

lemma *finite_fset_map: "finite (set_fmap (F :: ('a :: fintype) -~> 'b))"*
⟨*proof*⟩

lemma *rep_fmap_imp: "(! F x z . x ∉ dom (F :: ('a -~> 'b)) ⟶ P F ⟶ P (F (x ↦ z)))*
⟹ (! F x z . x ∉ fst ‘ (set_fmap F) ⟶ (pred_set_fmap P)(set_fmap F)
⟶ (pred_set_fmap P) (insert (x, z)(set_fmap F)))"

⟨*proof*⟩

Induction scheme for finite maps.

theorem *fmap_induct[rule_format]:*
"P (% x. None) ⟶ (! (F :: (('a :: fintype) -~> 'b)) x z . x ∉ dom F ⟶ P F ⟶ P (F (x ↦ z)))
⟶ P (F' :: ('a -~> 'b))"

⟨*proof*⟩

end

A.2 Basic Definition of the Sigma Calculus

theory *Sigma* **imports** *FiniteMaps* **begin**

The basic Sigma calculus, extended to include labeled terms for aspect-orientation. This formalisation uses de Bruijn indices for variables and was further extended to allow methods to have parameters.

Infrastructure for the finite maps

We use integers as method names, as they are the easiest to handle. To do so an arbitrary integer denotes the maximal number of different method names. Using strings with a fixed length would be also feasible.

consts *max_label :: nat*

axioms *LabelAvail: "max_label > 100"*

typedef *Label = "{n :: nat. n <= max_label }"*
⟨*proof*⟩

Finiteness of available labels.

lemma *finite_Label_set: "finite {n :: nat. n <= max_label }"*

⟨*proof*⟩

Translation between types.

lemma *Univ_abs_label: "(UNIV :: (Label set)) = Abs_Label ‘ {n :: nat. n <= max_label }"*
⟨*proof*⟩

Finiteness of the label universe.

lemma *finite_Label: "finite (UNIV :: (Label set))"*
⟨*proof*⟩

Labels are thus a finite type.

instance *Label :: "fintype"*
⟨*proof*⟩

Comparison of labels.

constdefs
Llt :: "[Label, Label] ⇒ bool" (**infixl** *"<" 50*)
"Llt a b == Rep_Label a < Rep_Label b"
Lle :: "[Label, Label] ⇒ bool" (**infixl** *"≤" 50*)
"Lle a b == Rep_Label a ≤ Rep_Label b"

Basic Datatype

The datatype realizing the syntax of the extended calculus. Types are introduced here to allow type annotations in ς terms. The variable type has cases for self variables and parameters.

datatype *type = Object "Label -~> (type × type × bool)"*

datatype *Variable = Self nat | Param nat*

datatype *dB =*
Var Variable
| Obj "Label -~> dB" type
| Call dB Label dB
| Upd dB Label dB
| Asp_Label nat dB ("_⟨_⟩")

Lifting and Substitution

De Bruijn indices add a few complications to the handling of variable substitution. These functions realize the required index adaption.

Lifting adjusts indices. The option cases are required for objects, which wrap a finite map.

```
primrec
 lift_Var ::     "[Variable,nat] ⇒ Variable"
where
   lift_Self:  "lift_Var (Self i) k = Self (if i < k then i else (i +
1))"
 | lift_Param: "lift_Var (Param i) k = Param (if i < k then i else (i
+ 1))"

primrec
  lift::"[dB, nat] ⇒ dB"
and
  lift_option::"[nat, dB option] ⇒ dB option"
where
   liftVar:    "lift (Var v) k = Var (lift_Var v k)"
 | liftCall:   "lift (Call a l b) k = Call (lift a k) l (lift b k)"
 | liftUpd:    "lift (Upd a l b) k = Upd (lift a k) l (lift b (k + 1))"

 | liftObj:    "lift (Obj f T) k = Obj (% l. (lift_option (Suc k) (f l)))
T"
 | liftLabel:  "(lift (i ⟨t⟩) k)  = (i ⟨(lift t k)⟩)"
 | lift_None:  "lift_option k None = None"
 | lift_Some:  "lift_option k (Some t) = Some (lift t k)"

primrec
  subst_Var ::    "[Variable, dB, dB, nat] ⇒ dB"
where
   subst_Self:  "subst_Var (Self i) a b k = (if k < i then Var (Self (i
- 1)) else if i = k then a else Var (Self i))"
   | subst_Param: "subst_Var (Param i) a b k = (if k < i then Var (Param
(i - 1)) else if i = k then b else Var (Param i))"
```

Substitution of variables.

```
primrec
  subst ::        "[dB, dB, dB, nat] ⇒ dB"  ("_[_,_'/_]" [300, 0, 0,
0] 300)
and
  subst_option :: "[nat, dB, dB, dB option] ⇒ dB option"
where
  subst_Var:     "(Var i)[s,t/k] = subst_Var i s t k"
  | subst_Call:  "(Call a l b)[s,t/k] = (Call (a[s,t/k]) l (b[s,t/k]))"
  | subst_Upd:   "Upd a l b [s,t/k] = (Upd (a [s,t/k]) l (b [lift s 0,
lift t 0 / k+1]))"
  | subst_Obj:   "Obj f T [s,t/k] = Obj (% l. (subst_option (Suc k)(lift
s 0)(lift t 0) (f l))) T"
  | subst_Label:  "(asp_la⟨x⟩) [s,t/k] = asp_la⟨(x[s,t/k])⟩"
  | subst_None:  "subst_option n s t None = None"
  | subst_Some:  "subst_option n s t (Some x) = Some (x [s,t/n])"

declare subst_Var [simp del]
```

Beta-reduction

The reduction relation for the calculus. This inductive definition realizes the semantics as small step operational semantics.

inductive *beta::"[dB, dB] => bool"* (**infixl** *"→β" 50*)
where
beta [simp, intro!] : "l : dom f ⟹ Call (Obj f T) l b →β (the(f l))[(Obj f T),b/0]"
| *upd [simp, intro!]: "l : dom f ⟹ Upd (Obj f T) l a →β Obj (f(l ↦ a)) T"*
| *sel [simp, intro!]: "s →β t ⟹ Call s l u →β Call t l u"*
| *selR [simp, intro!]: "u →β v ⟹ Call s l u →β Call s l v"*
| *updL [simp, intro!]: "s →β t ⟹ Upd s l u →β Upd t l u"*
| *updR [simp, intro!]: "s →β t ⟹ Upd u l s →β Upd u l t"*
| *obj [simp, intro!]: "⟦ s →β t; l: dom f ⟧ ⟹ Obj (f (l ↦ s)) T →β Obj (f (l ↦ t)) T"*
| *laba [simp, intro!]: "⟦s →β t ⟧ ⟹ i⟨s⟩ →β i⟨t⟩"*

inductive_cases *beta_cases [elim!]:*
"Var i →β t"
"Call s l t →β t"
"Upd s l t →β u"
"Obj s T →β Obj t T"
"i⟨s⟩ →β i⟨t⟩"

β^* is the transitive, reflexive closure of the reduction relation.

abbreviation
beta_reds :: "[dB, dB] => bool" (**infixl** *"->>" 50*) **where**
*"s ->> t == beta^** s t"*
abbreviation
beta_ascii :: "[dB, dB] => bool" (**infixl** *"->" 50*) **where**
"s -> t == beta s t"

notation *(latex)*
beta_reds (**infixl** *"→β* 50*)

declare *if_not_P [simp] not_less_eq [simp]*
— don't add *r_into_rtrancl[intro!]*

Congruence rules

A set of lemmas to allow the convenient use of β^* as a big step semantics.

lemma *rtrancl_beta_Obj [intro!]:*
"s →β s' ==> ((Call s l t) →β* (Call s' l t))"*
⟨*proof*⟩

lemma *rtrancl_beta_ObjR [intro!]:*
"s →β s' ==> ((Call t l s) →β* (Call t l s'))"* ⟨*proof*⟩

lemma *rtrancl_beta_updL:*
"s →β s' ⟹ Upd s l u →β* Upd s' l u"* ⟨*proof*⟩

lemma *rtrancl_beta_updR:*
"s →β s' ⟹ Upd u l s →β* Upd u l s'"* ⟨*proof*⟩

lemma *rtrancl_beta_upd:*
"⟦ u →β u'; s →β* s' ⟧ ⟹ Upd u l s →β* Upd u' l s'"*
⟨*proof*⟩

lemma *rtrancl_beta_obj:*
"⟦ s →β s'; l : dom f ⟧ ⟹ Obj (f (l ↦ s)) T →β* Obj (f (l ↦ s')) T"* ⟨*proof*⟩

lemma *rtrancl_beta_obj_no_dom:*
"⟦ s →β s' ⟧ ⟹ Obj (f (l ↦ s)) T →β* Obj (f (l ↦ s')) T"*
⟨*proof*⟩

lemma *rtrancl_beta_lab:*
"⟦ s →β s' ⟧ ⟹ i⟨(s)⟩ →β* i⟨(s')⟩ "* ⟨*proof*⟩

lemma *obj_lem: "⟦ n: dom f; the (f n) ->> x⟧ ⟹ Obj (f) T ->> Obj (f(n ↦ x)) T"*
⟨*proof*⟩

A helper function encoding the equality of submaps.

constdefs
Ltake_eq ::" [Label set, (Label ~=> 'a), (Label ~=> 'a)] ⇒ bool "
"Ltake_eq L f g == ∀l ∈ L . f l = g l"

Only the empty map has ∅ as domain.

lemma *empty_dom : " ⟦{} = dom g⟧ ⟹ g = empty"*
⟨*proof*⟩

If the submap's domain is the same as the domains of the compared maps, the maps are identical.

lemma *Ltake_eq_all : "⟦dom f = dom g; Ltake_eq (dom f) f g⟧ ⟹ f = g"*
⟨*proof*⟩

lemma *Ltake_eq_dom : "⟦ L ⊆ dom (f::(Label -~> 'a)) ; card L = card (dom f)⟧ ⟹ L = (dom f)"*
⟨*proof*⟩

lemma *Ltake_eq_empty [simp]: "Ltake_eq {} f g"*
⟨*proof*⟩

A major lemma for the transition from one Object to another. This very technical proof shows that one map(object) can reduce to another, one element(method) at a time.

lemma *rtrancl_beta_obj_lem0000:*
```
        "⟦ dom f = dom g; ∀ l ∈  dom  f. the (f l) ->> the (g l) ⟧ ⟹

        ∀  k  ≤ (card (dom f)) .
       (∃ ob . length ob = (k + 1) ∧
         (∀  obi. obi mem ob ⟶ dom (fst(obi)) =  dom f ∧
         ((snd obi)  ⊆ dom f)) ∧
         (fst (ob ! 0) = f)  ∧
         (card (snd (ob ! k)) = k) ∧
           (∀ i < k . snd (ob ! i ) ⊂ snd (ob ! k )) ∧
         (Obj (fst (ob ! 0)) T ->> Obj (fst (ob ! k)) T)  ∧
         ( card (snd (ob !k )) =  k ⟶ (Ltake_eq (snd (ob!k)) (fst (ob
! k)) g) ∧
                                        (Ltake_eq (( dom f) - (snd (ob!k)))
(fst (ob ! k)) f)


  ))"
```

⟨*proof*⟩
thm *rtrancl_trans*
⟨*proof*⟩

```
lemma rtrancl_beta_obj_n: "⟦ dom f = dom g; ∀  l ∈ dom f. the (f l)
->>  the (g l)⟧
                           ⟹ Obj f T ->> Obj g T"
```
⟨*proof*⟩
thm *Ltake_eq_all*
⟨*proof*⟩

Substitution-lemmas

Essential properties of the substitution function.

```
lemma subst_eq_s [simp]: "(Var (Self k))[u,t/k] = u"
```
⟨*proof*⟩

```
lemma subst_eq_p [simp]: "(Var (Param k))[u,t/k] = t"
```
⟨*proof*⟩

```
lemma subst_gt_p [simp]: "i < j ⟹ Var (Param j)[u,t/i] = Var (Param
(j - 1))"
```
⟨*proof*⟩

lemma *subst_gt_s [simp]: "i < j ⟹ Var (Self j)[u,t/i] = Var (Self (j - 1))"*
⟨*proof*⟩

lemma *subst_lt_p [simp]: "j < i ==> Var (Param j)[u,t/i] = Var (Param j)"*
⟨*proof*⟩

lemma *subst_lt_s [simp]: "j < i ==> Var (Self j)[u,t/i] = Var (Self j)"*
⟨*proof*⟩

Induction rules for fmaps and objects.

lemma *dB_induct: "⟦ !! n. P1 (Var n);*
!! l T. P1 (Obj l T);
!! d1 n d2.
⟦P1 d1; P1 d2⟧ ⟹ P1 (Call d1 n d2);
!! d1 n d2.
⟦P1 d1; P1 d2⟧
⟹ P1 (Upd d1 n d2);
!! d1 i. ⟦P1 d1⟧ ⟹ P1 (i⟨d1⟩)
⟧ ⟹ P1 dB"
⟨*proof*⟩

Idea: "P2 f" is actually just "! x. P1 (f x)" – instead of a searching for a general isomorphism between predicates on the ranfe of f, i.e. f x, and f itself – which doesn't exist, we instantiate the induction for the case that we need, i.e. P2 = P1 "but lifted to fmaps"

consts *foo:: " (dB ⇒ bool) ⇒ dB option ⇒ bool"*
primrec
f1: "foo P None = True"
f2: "foo P (Some x) = P x"

lemma *db_induct1:"⟦ ⋀n. P1 (Var n);*
⋀f. (! x: dom f. P1 (the (f x))) ⟹ P3 f;
⋀f T. P3 f ⟹ P1 (Obj f T);
⋀d1 l d2. ⟦P1 d1; P1 d2⟧ ⟹ P1 (Call d1 l d2);
⋀d1 l d2. ⟦P1 d1; P1 d2⟧ ⟹ P1 (Upd d1 l d2);
!! d1 i. ⟦P1 d1⟧ ⟹ P1 (i⟨d1⟩) ⟧
⟹ P1 db ∧ (P3 (f::Label -~>dB))"

⟨*proof*⟩

lemma *db_induct2:"⟦ ⋀n. P1 (Var n);*
⋀f T. P3 f ⟹ P1 (Obj f T);
⋀d1 l d2. ⟦P1 d1; P1 d2⟧ ⟹ P1 (Call d1 l d2);
⋀d1 l d2. ⟦P1 d1; P1 d2⟧ ⟹ P1 (Upd d1 l d2);
!! x. P3 (empty);
⋀ d1 f l . ⟦ l ∉ dom f; P1 d1; P3 f ⟧

```
                                ⟹ ( P3 (f(l ↦ d1)));
                    !! d1 i. ⟦P1 d1⟧ ⟹ P1 (i⟨d1⟩)⟧
                ⟹ P1 db ∧ (P3 (f::Label -~>dB ))"
```

⟨*proof*⟩

lemma *dB_induct2:*

```
"[| !!n. P (Var n); !!f T. Q f ==> P (Obj f T);
   !!d1 l d2.  [| P d1; P d2 |] ==> P (Call d1 l d2);
   !!d1 l d2. [| P d1; P d2 |] ==> P (Upd d1 l d2);
   !!x. Q empty;
   !!d1 f l. [| l ~: dom f; P d1; Q f |] ==> Q (f(l |-> d1));
   !!d1 i. P d1 ==> P (dB.Asp_Label i d1) |]
==> P db & Q f"
```

⟨*proof*⟩

More lifting

Essential properties of the lifting function, especially commutative properties and in the combination with substitution.

lemma *lift_lift_prep [rule_format]:*

```
    "   (∀i k   . i < k + 1 --> lift (lift t i) (Suc k) = lift (lift
t k) i)
    &  ( ∀i k   .  i < k + 1  --> (lift (lift (Obj fun T) i) (Suc k)
= lift (lift (Obj fun T) k) i))
"
```

⟨*proof*⟩

lemma *lift_lift [rule_format]:*

```
    "(∀i k. i < k + 1 --> lift (lift t i) (Suc k) = lift (lift t k) i)"
```

⟨*proof*⟩

lemma *lift_subst_prep :*

```
    "(∀i j s t. j < i + 1 --> lift (a[s,t/j]) i = (lift a (i + 1)) [lift
s i,lift t i / j])
    &(∀i j s t. j < i + 1 --> lift (Obj f T [s,t/j]) i = (lift (Obj f
T) (i + 1)) [lift s i,lift t i / j])"
```

⟨*proof*⟩

lemma *lift_subst [simp]:*

```
    " j < i + 1 --> lift (t[s,u/j]) i = (lift t (i + 1)) [lift s i,lift
u i / j]"
```

⟨*proof*⟩

lemma *lift_subst_lt_prep:*

```
    "(∀i j s u. i < j + 1 --> lift (t[s,u/j]) i = (lift t i) [lift s
i,lift u i / j + 1])
```

```
    &(∀ i j s u. i < j + 1 --> lift ((Obj f T)[s,u/j]) i = (lift (Obj
f T) i) [lift s i,lift u i / j + 1])"
```
⟨*proof*⟩

lemma *lift_subst_lt:*
```
    " i < Suc j ⟹ lift (t[s,u/j]) i = (lift t i) [lift s i,lift u i
/ j + 1]"
```
⟨*proof*⟩

lemma *subst_lift_prep :*
```
    "(∀ k s u. (lift t k)[s,u/k] = t) & (∀ k s u. (lift (Obj f T) k)[s,u/k]
= Obj f T)"
```
⟨*proof*⟩

lemma *subst_lift [simp]:*
```
    "∀ k s u. (lift t k)[s,u/k] = t"
```
⟨*proof*⟩

lemma *subst_subst_prep [rule_format]:*
```
    "(∀ i j u v s s'. i < j + 1 ⟶
                ((t[(lift v i),(lift s i) / Suc j])[u[v,s/j],s'[v,s/j]/i]
= (t[u,s'/i])[v,s/j]))
    &(∀ i j u v s s'. (i < j + 1 ⟶
                      (Obj f T [(lift v i),(lift s i) / Suc j])[u[v,s/j],s'[v,s/j]/i]
= (Obj f T[u,s'/i])[v,s/j]))"
```
⟨*proof*⟩

lemma *subst_subst [rule_format]:*
```
    " i < j + 1 ⟹ t[lift v i,lift s' i / Suc j][u[v,s'/j],s[v,s'/j]/i]
= t[u,s/i][v,s'/j]"
```
⟨*proof*⟩

lemma *rmp:* `"⟦ A; A ⟶ B ⟧ ⟹ B"`
⟨*proof*⟩

lemma *insert_select [simp] :* `"!!f l t. the ((f(l ↦ t)) l) = t"`
⟨*proof*⟩

lemma *dom_insert [simp] :* `"!!f l t. ⟦ l ∈ dom f ⟧ ⟹ dom (f(l ↦ t))
= dom f"`
⟨*proof*⟩

lemma *insert_select2 [simp] :* `"!!f l1 l2 t.⟦l1 ≠ l2⟧ ⟹ ((f(l1 ↦
t)) l2) = (f l2)"`
⟨*proof*⟩

lemma *the_insert_select [simp] :* `"!!f l1 l2 t.⟦l2 ∈ dom f; l1 ≠ l2⟧
⟹ the ((f(l1 ↦ t)) l2) = the (f l2)"`
⟨*proof*⟩

end

A.3 Confluence of the Reduction

theory *Confluence* **imports** *Sigma Commutation* **begin**

Parallel Reduction

This confluence proof is based on the framework by Tobias Nipkow. The essential idea is to design a parallel reduction relation for which confluence can be shown. By establishing that the relation is between the normal reduction and its transitive, reflexive closure, the result can be applied to the original reduction.

inductive *par_beta::"[dB, dB] => bool"* (**infixl** *"=>" 50*)
where

```
    var [simp, intro!]: "Var n => Var n"
    |obj [simp, intro!]: "⟦dom s = dom s'; ∀  l ∈ dom s . the (s l) =>
the (s' l) ⟧ ⟹ Obj s B => Obj s' B"
    |upd [simp, intro!]: "⟦s => s'; t => t' ⟧ ⟹ Upd s l t => Upd s'
l t'"
    |upd' [simp, intro!]:"⟦Obj s B => Obj s' B; t => t'; l ∈ dom s ⟧
                         ⟹(Upd (Obj s B) l t)  => (Obj (s'(l ↦ t'))
B)"
    |sel [simp, intro!]: "⟦s => t; u => v ⟧ ⟹ Call s l u => Call t l
v"
    |beta [simp, intro!]: "⟦Obj f B => Obj f' B; l ∈ dom f'; b => b' ⟧
⟹ Call (Obj f B) l b => (the (f' l))[(Obj f' B),b'/0]"
    |lab' [simp, intro!]: "⟦ t => t'⟧ ⟹ i⟨t⟩ => i⟨t'⟩"
```

inductive_cases *par_beta_cases [elim!]:*

```
  "Var n => t"
  "Obj s B => t"
  "Call s l b => t"
  "Upd s l t => u"
  "i⟨t⟩ => i⟨t⟩"
```

Inclusions

*beta ⊆ par_beta ⊆ beta^**

Variables do not reduce

lemma *par_beta_varL [simp]:*

```
  "(Var n => t) = (t = Var n)"
```
⟨*proof*⟩

Parallel reduction in objects.

lemma *par_beta_obj_subst': "⟦ l ∈ dom f; s => t; Obj f B => Obj f' B ⟧*
⟹ Obj (f (l ↦ s)) B=> Obj (f'(l ↦ t)) B"

⟨*proof*⟩

Parallel reduction in objects, stronger version.

lemma *par_beta_obj_subst: "⟦ s => t; Obj f B => Obj f' B ⟧*
⟹ Obj (f (l ↦ s)) B => Obj (f'(l ↦ t)) B"

⟨*proof*⟩

lemma *par_beta_refl_prep: " t => t & Obj l T => Obj l T"*
⟨*proof*⟩

Parallel reduction is reflexive.

lemma *par_beta_refl [simp]: "t => t"*

⟨*proof*⟩

Parallel reduction is bigger (or equal) than the original reduction.

lemma *beta_subset_par_beta: "beta <= par_beta"*
⟨*proof*⟩

Parallel reduction is smaller than the transitive, reflexive closure of the original reduction.

lemma *par_beta_subset_beta: "par_beta <= beta^**"*
⟨*proof*⟩

Inversion of par_beta.

lemma *Obj_par_red: "⟦ Obj s T=> z ⟧ ⟹ ? lz. dom s = dom lz & z = Obj lz T"*
⟨*proof*⟩

lemma *Upd_par_red: "⟦ Upd s l t => z ⟧ ⟹*
(? s' t'. s => s' & t => t' & z = Upd s' l t') |
(? f f' t' T. s = Obj f T & Obj f T => Obj f' T & t => t' & z = Obj (f' (l ↦ t')) T)"

⟨*proof*⟩

lemma *Call_par_red: "⟦ Call s l t => z ⟧ ⟹*
(? s' t'. s => s' & t => t' & z = Call s' l t')
| (? f f' T t' . Obj f T => Obj f' T & l ∈ dom f' & s = Obj f T
& t => t' & z = (the (f' l))[(Obj f' T),t'/0])"

⟨*proof*⟩

Misc properties of par_beta and domains

A number of helper lemmata for the later proofs.

lemma *lift_obj_lem0: "Obj (f) T => t ⟹ ∃ f'. dom f = dom f' & t = Obj (f') T"*
⟨*proof*⟩

lemma *lift_obj_lem00: "Obj (f) T=> Obj (f') T ⟹ dom f = dom f'"*
⟨*proof*⟩

lemma *dom_liftoption_lem: " !!f . dom f = dom (λ l. lift_option n (f l))"*
⟨*proof*⟩

lemma *insert_dom_eq : "⟦dom f = dom f'⟧ ⟹ dom (f(l |-> x)) = dom (f'(l ↦ x'))"*
⟨*proof*⟩

lemma *insert_dom_less_eq : "!! y y' x . ⟦x ∉ dom f; x ∉ dom f'; dom (f(x ↦ y)) = dom (f'(x ↦ y'))⟧ ⟹ dom f = dom f'"*
⟨*proof*⟩

If one object reduces to another, then any shared element can be removed from both objects without harming the reduction.

lemma *par_beta_less : "!! d1 t1 . ⟦ Obj (f(l↦d1)) T => Obj(t(l |-> t1)) T ; l ∉ dom f; l ∉ dom t ⟧ ⟹ Obj f T => Obj t T"*
⟨*proof*⟩

lemma *par_beta_less' : "!! d1 t1 . ⟦ ∀ x ∈ dom f. the ((f(l↦d1)) x) => the ((t(l |-> t1)) x) ; l ∉ dom f; l ∉ dom t ⟧ ⟹ ∀ x ∈ dom f . the (f x) => the (t x)"*
⟨*proof*⟩

If two objects are in a reduction relation, then all their elements are as well.

lemma *par_beta_one : "!! d1 t1 . ⟦ Obj (f(l↦d1)) T => Obj(t(l |-> t1)) T ; l ∉ dom f; l ∉ dom t ⟧ ⟹ d1 => t1"*
⟨*proof*⟩

lemma *obj_par_beta : "⟦Obj f T => Obj f' T⟧ ⟹ ∀ l∈ dom f . the (f l) => the (f' l)"*

⟨*proof*⟩

Lifting maintains the parallel reduction

lemma *lift_obj_lem: "⟦ lift (Obj f T) n => lift (Obj f' T) n; lift x (Suc n) => lift x' (Suc n)⟧ ⟹ lift (Obj (f(l ↦ x)) T) n => lift (Obj (f'(l |-> x')) T) n"*

⟨*proof*⟩

lemma *dom_substoption_lem: "dom (λ l . subst_option n s t (f l)) = dom f"*
⟨*proof*⟩

lemma *olift_lift: "! n'. n' ∈ dom f ⟶ (the ((λ l . lift_option n (f l)) n')) = (lift (the (f n')) n)"*
⟨*proof*⟩

lemma *olift_upd: "! n'. ((λ l . lift_option (Suc n) ((s(n'|-> t)) l)) = ((λ l . lift_option (Suc n)(s l))(n' ↦ (lift t (Suc n)))))"*
⟨*proof*⟩

Helper; beta is a subset of par_beta.

lemma *par_beta_beta : "!!a b. ⟦ a -> b⟧ ⟹ a => b"*

⟨*proof*⟩

There is always an object with one element removed that is identical for all other elements.

lemma *insert_one : "⟦l∈ dom t⟧ ⟹ ∃ t' . l ∉ dom t' ∧ t = (t'(l ↦ (the (t l))))"*
⟨*proof*⟩

lemma *obj_par_beta' : "⟦ ∀l∈ dom f . the (f l) => the (f' l); dom f = dom f'⟧ ⟹ Obj f T => Obj f' T"*
⟨*proof*⟩

lemma *lab_par_beta : "⟦dB.Asp_Label i d1 => t' ⟧ ⟹ (∃ t''. t' = dB.Asp_Label i t'' ∧ d1 => t'')"*
⟨*proof*⟩

Parallel reduction does not change type annotations.

lemma *par_red_type_obj [simp] : "Obj f T => Obj f' T' ⟹ T = T'"*
⟨*proof*⟩

Major lemma: lifting maintains parallel reduction.

lemma *par_beta_lift_prep [rule_format, simp]:*
```
    "(∀t' n. t => t' --> lift t n => lift t' n)
    &(∀t' n T. Obj f T => Obj t' T --> lift (Obj f T) n => lift (Obj
t' T) n)"
```
⟨*proof*⟩

Major lemma: substitution maintains parallel reduction.

lemma *par_beta_subst_prep [rule_format]:*
```
    "(∀s s' t' b b' n. s => s' ∧ b => b' --> t => t' --> t[s,b/n] =>
t'[s',b'/n])
    &(∀s s' t' b b' n T. s => s' ∧ b => b' --> (Obj f T) => t' --> Obj
f T[s,b/n]  => t'[s',b'/n])"
```
⟨*proof*⟩
thm *par_beta_less*
⟨*proof*⟩

lemma *par_beta_subst [rule_format]:*
```
    "∀s s' t' n. s => s' ∧ b => b' --> t => t' --> t[s,b/n] => t'[s',b'/n]"
```
⟨*proof*⟩

There is always a smaller object.

lemma *one_more_dom :"!! f . ⟦f ≠ empty⟧ ⟹ ∀ l ∈ dom f . ∃ f' . f
= f'(l ↦ the(f l)) ∧ l ∉ dom f' "*
⟨*proof*⟩

lemma *fmap_inducta: " ⟦P empty;*
```
                !! x (F::Label -~>'a) y . ⟦  P F; x ∉ dom F ⟧
                ⟹ P (F(x ↦ y))  ⟧ ⟹ P F"
```
⟨*proof*⟩

lemma *fmap_induct3: "!! (F2::Label -~> 'b) (F3::Label -~> 'c) . ⟦
dom (F1::Label -~> 'a) = dom F2; dom F3 = dom F1; P empty empty empty;*
```
                !! x a b c (F1::Label -~> 'a)  (F2::Label -~> 'b)  (F3::Label
-~> 'c)  . ⟦  P F1 F2 F3; dom F1 = dom F2; dom F3 = dom F1;  x ∉ dom F1
⟧
                ⟹ P (F1(x ↦ a)) (F2(x ↦ b)) (F3(x ↦ c))  ⟧ ⟹ P
F1 F2 F3"
```

⟨*proof*⟩

lemma *fmap_induct2: "!! (F2::Label -~> 'b) . ⟦ dom (F1::Label -~>
'a) = dom F2; P empty empty;*
```
                !! x a b  (F1::Label -~> 'a)  (F2::Label -~> 'b) . ⟦  P
F1 F2 ; dom F1 = dom F2;  x ∉ dom F1 ⟧
```

```
                ⟹ P (F1(x ↦ a)) (F2(x ↦ b))  ⟧ ⟹ P F1 F2"
```

⟨*proof*⟩

Very technical transformation of a !? into a ?! for lists and par_red.

lemma *ex_list_ex0: " ⟦ dom (s::Label -~>dB) = dom (s'::Label -~>dB) ; dom (lz::Label -~>dB) = dom s ⟧ ⟹*
(∀ l ∈ dom s . the (s l) => the (lz l))
& (∀ l ∈ dom s. (! z. the (s l) => z ⟶ (? u. the (s' l) => u ∧ z => u)))
⟶ (∃ lu. (dom lu = dom s) & (∀ l ∈ dom s. the (s' l) => the (lu l) ∧ the (lz l) => the (lu l)))"
⟨*proof*⟩

lemma *ex_list_ex: "⟦ dom (s::Label -~>dB) = dom (s'::Label -~>dB); dom s = dom (lz::Label -~>dB); ∀ l ∈ dom s. the (s l) => the (lz l);*
(∀ l ∈ dom s. (∀z. the (s l) => z ⟶ (∃u. the (s' l) => u ∧ z => u))) ⟧
⟹ (∃ lu. dom lu = dom s ∧ (∀ l ∈ dom s. the (s' l) => the (lu l) ∧ the (lz l) => the (lu l)))"
⟨*proof*⟩

Confluence (directly)

Main result: Confluence of beta relation for Sigma calculus by diamond property of parallel reduction and $\beta \subseteq par\_beta \subseteq \beta^*$

lemma *diamond_par_beta: "diamond par_beta"*
⟨*proof*⟩

Confluence (classical not via complete developments)

Main theorem: Confluence of β

theorem *beta_confluent: "confluent beta"*
⟨*proof*⟩

end

A.4 First Order Types for Sigma Terms with Aspect Labels

theory *Environments* **imports** *"../coreCalculus/Preliminaries"* **begin**

Type Environments

Some basic properties of our list/stack variable environments. The position in the list denotes the distance to the binder.

datatype *('a) environment = Env "('a × 'a) list"*

Adding variables.

primrec
shift :: "('a environment) ⇒ nat ⇒ 'a ⇒ 'a
⇒ 'a environment" ("_<_:_,_>" [90, 50, 0, 0] 91)
where
shift_def [simp]: "(Env e)<i:a,b> = Env (list_insert e i (a,b))"

Retrieving variables.

primrec
env_at :: "('a environment) ⇒ nat ⇒ ('a × 'a) " ("_!_")
where
env_at_def [simp]: "(Env e)!i = e!i"

Size of the environment (also establishes smallest free variable).

primrec
env_length :: "('a environment) ⇒ nat"
where
env_length_def [simp]: "env_length (Env e) = length e"

syntax *(latex* **output***)*
env_length :: "('a environment) ⇒ nat" ("|_|" 91)

notation *(latex) env_length ("|_|" 60)*

constdefs
prefix :: "('a list) ⇒ ('a list) ⇒ bool"
"prefix l m == l = take (length l) m"

syntax *(xsymbols)*
shift :: "('a environment) ⇒ nat ⇒ 'a ⇒ 'a ⇒ 'a list" ("_⟨_:_,_⟩"
[90, 0, 0] 91)
syntax *(HTML* **output***)*
shift :: "('a environment) ⇒ nat ⇒ 'a ⇒ 'a ⇒ 'a list" ("_⟨_:_,_⟩"
[90, 0, 0] 91)

lemma *shift_eq [simp]: "i = j ∧ j ≤ env_length e ⟹ (e⟨i:T,U⟩)!(j)*
= (T,U)"
⟨*proof*⟩

lemma *shift_gt [simp]: "⟦j < i; i ≤ env_length e⟧ ⟹ (e⟨i:T,U⟩)! j = e!j"*
⟨*proof*⟩

lemma *shift_lt [simp]: "⟦i ≤ j⟧ ⟹ (e⟨i:T,U⟩)! (Suc j) = e!(j)"*
⟨*proof*⟩

lemma *shift_lt2 [simp]: "⟦i < j; i ≤ env_length e⟧ ⟹ (e⟨i:T,U⟩)!(j) = e!(j - 1)"*
⟨*proof*⟩

lemma *shift_lt3: "⟦i < j⟧ ⟹ (e⟨i:T,U⟩)!Suc(j - 1) = e!(j - 1)"*
⟨*proof*⟩

lemma *shift_lt4 [simp]: "⟦i < j⟧ ⟹ (e⟨i:T,U⟩)!(j) = e!(j - 1)"*
⟨*proof*⟩

lemma *shift__eq_same [simp]: "⟦i ≠ j; i <= env_length e; j <= env_length e ⟧ ⟹ (e⟨i:A,T⟩)!(j) = (e⟨i:B,U⟩)!(j) "*
⟨*proof*⟩
thm *shift_gt*
⟨*proof*⟩

lemma *shift_commute [simp]: "⟦i ≤ env_length e⟧ ⟹ e⟨i:U,V⟩⟨0:T,W⟩ = e⟨0:T,W⟩⟨Suc i:U,V⟩"* ⟨*proof*⟩

lemma *shift_length [simp] :*
"∀ x. env_length (e⟨x:t,u⟩) = Suc (env_length e)"
⟨*proof*⟩

Retrieving the self variable with a given index.

primrec
env_self_at :: "('a environment) ⇒ nat ⇒ ('a) " ("_#_")
where
env_self_at_def [simp]: "(Env e)#i = fst(e!i)"

Retrieving the parameter variable with a given index.

primrec
env_param_at :: "('a environment) ⇒ nat ⇒ ('a) " ("_§_")
where
env_param_at_def [simp]: "(Env e)§i = snd(e!i)"

Re-establishing properties for self/parameter.

lemma *shift_eq_self [simp]: "i = j ∧ j ≤ env_length e ⟹ (e⟨i:T,U⟩)#(j) = (T)"*
⟨*proof*⟩

lemma *shift_eq_param [simp]:* "i = j ∧ j ≤ env_length e ⟹ (e⟨i:T,U⟩)§(j) = (U)"
⟨*proof*⟩

lemma *shift_gt_self [simp]:* "⟦j < i; i ≤ env_length e⟧ ⟹ (e⟨i:T,U⟩)#j = e#j"
⟨*proof*⟩

lemma *shift_gt_param [simp]:* "⟦j < i; i ≤ env_length e⟧ ⟹ (e⟨i:T,U⟩)§j = e§j"
⟨*proof*⟩

lemma *shift_lt_self [simp]:* "⟦i ≤ j⟧ ⟹ (e⟨i:T,U⟩)#(Suc j) = e#(j)"
⟨*proof*⟩

lemma *shift_lt_param [simp]:* "⟦i ≤ j⟧ ⟹ (e⟨i:T,U⟩)§(Suc j) = e§(j)"
⟨*proof*⟩

lemma *shift_lt2_self [simp]:* "⟦i < j; i ≤ env_length e⟧ ⟹ (e⟨i:T,U⟩)#(j) = e#(j - 1)"
⟨*proof*⟩

lemma *shift_lt2_param [simp]:* "⟦i < j; i ≤ env_length e⟧ ⟹ (e⟨i:T,U⟩)§(j) = e§(j - 1)"
⟨*proof*⟩

lemma *shift_lt3_self:* "⟦i < j⟧ ⟹ (e⟨i:T,U⟩)#Suc(j - 1) = e#(j - 1)"
⟨*proof*⟩

lemma *shift_lt3_param:* "⟦i < j⟧ ⟹ (e⟨i:T,U⟩)§Suc(j - 1) = e§(j - 1)"
⟨*proof*⟩

lemma *shift_lt4_self [simp]:* "⟦i < j⟧ ⟹ (e⟨i:T,U⟩)#(j) = e#(j - 1)"
⟨*proof*⟩

lemma *shift_lt4_param [simp]:* "⟦i < j⟧ ⟹ (e⟨i:T,U⟩)§(j) = e§(j - 1)"
⟨*proof*⟩

lemma *shift__eq_same_self [simp]:* "⟦i ≠ j; i <= env_length e; j <= env_length e ⟧ ⟹ (e⟨i:A,T⟩)#(j) = (e⟨i:B,U⟩)#(j) "
⟨*proof*⟩

lemma *shift__eq_same_param [simp]:* "⟦i ≠ j; i <= env_length e; j <= env_length e ⟧ ⟹ (e⟨i:A,T⟩)§(j) = (e⟨i:B,U⟩)§(j) "
⟨*proof*⟩

Additional properties of Types and Environments

Proof-technical additions needed for later reasoning about variable environments

primrec
```
   strp :: "('a environment)  ⇒ (('a × 'a)  list)"
```
where
```
  strp_def: "(strp (Env a)) = a"
```

primrec
```
   env_app :: "('a environment) ⇒  ('a environment) ⇒ ('a environment)"
("_+_")
```
where
```
  env_app_def: "env_app (Env a) b =  Env (a@(strp b))"
```

lemma `shift_append' : "(e+(Env [x]))<0:X,Y> = (e<0:X,Y>)+Env [x]"`
⟨*proof*⟩

lemma `shift_append : "(env+x)<0:X,Y> =(env<0:X,Y>)+x"`
⟨*proof*⟩

lemma `prefix_implies :  "⟦prefix l m ⟧ ⟹ ∀ x < length l . l!x = m!x"`

⟨*proof*⟩

lemma `take_nth' :"⟦  a < length xs  ⟧ ⟹ (xs  ! a) = ((xs@[x]) ! a)"`
⟨*proof*⟩

lemma `empty_prefix : "prefix [] l"` ⟨*proof*⟩

lemma `shift_prepend  : "(e<0:a,b>) = ((Env [(a,b)]) + (e))"`
⟨*proof*⟩

lemma `shift_prepend' : "(env<0:a,b>+x) = (env+x)<0:a,b>"`
⟨*proof*⟩

lemma `env_length_add [simp]:`
```
        "env_length (a + b) = env_length a + env_length b "
```
⟨*proof*⟩

lemma `env_length_leq [simp]:" env_length (a) ≤ (env_length ( a + b))"`

⟨*proof*⟩

lemma `nth_prefix : " ⟦prefix l m; x < length l⟧ ⟹ l!x = m!x"`
⟨*proof*⟩

lemma `prefix_length_trans: " ⟦x < length l; prefix l l' ⟧ ⟹ x < length l'"`

⟨*proof*⟩

lemma *nth_append_self: "(xs + ys) # n = (if n < env_length xs then xs # n else ys # (n - env_length xs))"*
⟨*proof*⟩

lemma *nth_append_param: "(xs + ys) § n = (if n < env_length xs then xs § n else ys § (n - env_length xs))"*
⟨*proof*⟩

end

theory *SubTypes* **imports** *"../coreCalculus/Sigma"* **begin**

Subtypes

Definition of the subtype relation for depth subtyping

Function to lift dom to types.

```
primrec
  do :: "type ⇒ (Label set)"
where
 "do (Object l) = (dom l)"
```

Retrieving a method type.

```
primrec
  type_get :: "type ⇒ Label ⇒ (type × type × bool) option "  ("_^_"
1000)
where
 " (Object l)^n =  (l n) "
```

Helpers to avoid definedness issues.

```
primrec
  fst_opt :: "(type*type × bool) option ⇒ type option"
where
 " fst_opt (Some Y) = Some (fst Y)"
 |" fst_opt None = None"
```

```
primrec
  snd_opt :: "(type × type × bool) option ⇒ type option"
where
 " snd_opt (Some Y) = Some (fst (snd Y))"
```

```
 |" snd_opt None = None"
```

primrec
```
  variance :: "(type × type × bool) option ⇒ (bool option)"
```
where
```
 " variance (Some Y) = (Some (snd (snd Y)))"
 |" variance None = None"
```

Helper to accommodate width subtyping.

fun
```
  var_imp :: "(bool) option ⇒ (bool) option ⇒ (bool)"
```
where
```
 "  var_imp (Some Y) (Some X) = (Y ⟶ X)"
 |" var_imp None x = False"
 |" var_imp x None = False"
```

lemma *var_imp_Some1 : "⟦(var_imp (f) (g)) ⟧ ⟹ ∃ a. f = Some a"*
⟨*proof*⟩

lemma *var_imp_Some2 : "⟦(var_imp (f) (g)) ⟧ ⟹ ∃ a. g = Some a"*
⟨*proof*⟩

lemma *var_imp_dom1 : "⟦(var_imp (f x) (g x)) ⟧ ⟹ x ∈ dom f"*
⟨*proof*⟩

lemma *var_imp_dom2 : "⟦(var_imp (f x) (g x)) ⟧ ⟹ x ∈ dom g"*
⟨*proof*⟩

lemma *var_imp_imp :" ⟦var_imp (f x) (g x)⟧ ⟹ the (f x) ⟶ the (g x)"*
⟨*proof*⟩

lemma *var_imp_imp' :" ⟦var_imp (f) (g)⟧ ⟹ the (f) ⟶ the (g)"*
⟨*proof*⟩

Inductive definition of the subtype relation

inductive *sig_subtype:: "type ⇒ type ⇒ bool"   (" _ <: _" 280)*
where

```
    sub_obj [ intro!]:     "⟦∀ l ∈ (do B). (
                              (
                                ((the (variance (B^l))) ∧ (((the (fst_opt
(A^l)))) <: ((the (fst_opt (B^l))))) ∧   ((the (snd_opt (B^l))) <: (the
(snd_opt (A^l)))) )
                                  ∨
                                ((((fst_opt (B^l))) = ((fst_opt (A^l))))
∧  ((snd_opt (B^l)) = ((snd_opt (A^l)))))
```

```
                                )
                                  ∧  (var_imp (variance (A^l)) (variance
(B^l)))
                               )⟧ ⟹ A <: B"
  |subtype_refl [simp, intro!]:     " A <: A"
  |subtype_trans:  " ⟦A <: B; B <: C⟧ ⟹  A <: C"
```

The empty object is the shared supertype of all types.

theorem *subtype_top: "A <: Object empty"*
⟨*proof*⟩

Definedness rules.

lemma *fst_in_dom: " ⟦fst_opt (f a) = Some y ⟧ ⟹ a ∈ dom f"*
⟨*proof*⟩

lemma *snd_in_dom: " ⟦snd_opt (f a) = Some y ⟧ ⟹ a ∈ dom f"*
⟨*proof*⟩

lemma *fst_opt_some: " ⟦ a ∈ dom f ⟧ ⟹ ∃ y. fst_opt (f a) = Some y"*
⟨*proof*⟩

lemma *snd_opt_some: " ⟦ a ∈ dom f ⟧ ⟹ ∃ y. snd_opt (f a) = Some y"*
⟨*proof*⟩

lemma *var_imp_var_dom1 : "⟦var_imp (variance (f x)) (variance (funa x))*
⟧ ⟹ x ∈ dom f"
⟨*proof*⟩

lemma *var_imp_var_dom2 : "⟦var_imp (variance (f x)) (variance (funa x))*
⟧ ⟹ x ∈ dom funa"
⟨*proof*⟩

A subtype shares all methods of a supertype.

lemma *dom_subset: " ⟦A <: B ⟧ ⟹ do B ⊆ do A"*
⟨*proof*⟩

Equality rules.

lemma *opt_snd_eq :"⟦the (snd_opt A^l) = the (snd_opt B^l); l ∈ do A;*
l ∈ do B⟧ ⟹ snd_opt A^l = snd_opt B^l"
⟨*proof*⟩

lemma *opt_snd_eq' :"⟦l ∈ do A⟧ ⟹ the (snd_opt A^l) = (fst (snd (the*
A^l)))"
⟨*proof*⟩

lemma *opt_fst_eq :"⟦the (fst_opt A^l) = the (fst_opt B^l); l ∈ do A;*
l ∈ do B⟧ ⟹ fst_opt A^l = fst_opt B^l"

⟨*proof*⟩

lemma *opt_fst_eq' :"⟦l ∈ do A⟧ ⟹ the (fst_opt A^l) = (fst (the A^l))"*
⟨*proof*⟩

lemma *opt_fst_eq'' :"⟦a ∈ dom f⟧ ⟹ the (fst_opt (f a)) = (fst (the (f a)))"*
⟨*proof*⟩

lemma *var_the :"⟦(the (variance ((A::type)^l))); l ∈ do A⟧ ⟹ (variance (A^l)) = (Some True)"*
⟨*proof*⟩

lemma *var_the_not :"⟦¬(the (variance ((A::type)^l))); l ∈ do A⟧ ⟹ (variance (A^l)) = (Some False)"*
⟨*proof*⟩

Inversion lemma for return types.

lemma *sub_eq: " ⟦A <: B⟧ ⟹ ∀ l ∈ do B . fst (the A^l) <: fst (the B^l) "*
⟨*proof*⟩

Inversion lemma for parameter types.

lemma *sub_sub: " ⟦A <: B ⟧ ⟹ ∀ l ∈ do B . the (snd_opt (B^l)) <: the (snd_opt (A^l))"*
⟨*proof*⟩

Stronger inversion lemma.

lemma *sub_inv : "⟦A <: B⟧ ⟹ ∀ l ∈ (do B). ((((the (fst_opt (A^l)))) <: ((the (fst_opt (B^l))))) ∧ ((the (snd_opt (B^l))) <: (the (snd_opt (A^l)))) ∧ (the (variance (A^l)) ⟶ (the (variance (B^l)))))"*
⟨*proof*⟩

Strongest inversion lemma.

lemma *sub_inv2 : "⟦A <: B⟧ ⟹*
∀ l ∈ (do B).
(((the (variance (B^l))) ∧ ((((the (fst_opt (A^l))) <: (the (fst_opt (B^l))))
∧ (the (snd_opt (B^l))) <: (the (snd_opt (A^l))))
)
∨
(((the (fst_opt (A^l))) = (the (fst_opt (B^l)))) ∧ ((the (snd_opt (B^l))) = (the (snd_opt (A^l)))))))
∧ (var_imp (variance (A^l)) (variance (B^l)))"
⟨*proof*⟩
thm *var_imp_imp*
⟨*proof*⟩

Equality of types.

lemma *type_eq :"⟦∀ l ∈ do A. A^l = B^l; do A = do B⟧ ⟹ A = B"*
⟨*proof*⟩

Anti-symmetry of the domains of related types.

lemma *dom_antisym : "⟦A <: B; B <: A⟧ ⟹ do A = do B"*
⟨*proof*⟩

lemma *type_eq': "⟦fst_opt (A^l) = fst_opt (B^l); snd_opt (A^l) = snd_opt (B^l); (variance (A^l)) = ((variance (B^l)))⟧ ⟹ A^l = B^l"*
⟨*proof*⟩
thm *fst_in_dom*
⟨*proof*⟩

lemma *sub_antisym' : "⟦ A <: B⟧ ⟹ (B <: A ⟶ A = B)"*

⟨*proof*⟩
thm *empty_dom*
⟨*proof*⟩

Anti-symmetry of the relation.

lemma *sub_antisym : "⟦ A <: B; B <: A ⟧ ⟹ A = B"* ⟨*proof*⟩
end

theory *TypedSigma* **imports** *Environments SubTypes* **begin**

Types and Typing Rules

The inductive definition of the typing relation.

constdefs
param :: "(type × type × bool) ⇒ type " "param a == fst (snd a)"

inductive *typing::"(type environment) ⇒ (Sigma.type list) ⇒ dB ⇒ type ⇒ bool" ("_,_⊢ _ : _" [80, 0, 80] 230)*
where
T_Var [intro!]: "⟦(x::nat) < (env_length env); (env#x) = T; T <: A ⟧ ⟹ env,L ⊢ Var (Self x) : A"

```
  | T_Param [ intro!]:    "⟦(x::nat) < (env_length env); (env§x) = T;
T <: A ⟧ ⟹ env,L ⊢ Var (Param x) : A"
  | T_Obj  [ intro!]:     "⟦ dom b = do B; ∀l ∈ (do B) . (env<0:B,(param(the
(B^l)))>,L ⊢ (the (b l)) : fst((the (B^l)))); B <: A  ⟧ ⟹ env,L ⊢ (Obj
b B) :  A"
  | T_Upd  [ intro!]:     "⟦ env,L ⊢ a : A; ¬(the (variance (A^l)));
l ∈ do A ; env<0:A,param(the (A^l))>,L ⊢ n : fst(the (A^l)); A <: B ⟧
⟹ env,L ⊢  (Upd a l n) : B"
  | T_Call [ intro!]:     "⟦ env,L ⊢ a : A ; env,L ⊢ b : param(the (A^l))
; l ∈ do A; fst(the (A^l)) <: B ⟧ ⟹ env,L ⊢ (Call a l b)  : B"
  | T_lab  [ intro!]:     "⟦ env,L ⊢ a : A; i < length L; L!i = A; A
<: B⟧ ⟹ env,L ⊢ i⟨a⟩  : B"


inductive_cases typing_elims [elim!]:
  "e,L ⊢ Obj b T : T"
  "e,L ⊢ Var (Self i) : T"
  "e,L ⊢ Var (Param i) : T"
  "e,L ⊢ Call a l b : T"
  "e,L ⊢ Upd a l n  : T"
  "e,L ⊢ i⟨t⟩ : T"
```

Basic lemmas

Basic traits of the type system.

A term having a subtype also has any supertype.

lemma *subsumption [intro!]*: "⟦e,L ⊢ t : T; T <: A⟧ ⟹ e,L ⊢ t : A"
⟨*proof*⟩

Inversion rules for objects.

lemma *obj_inv'* : "e,L ⊢ Obj f U : A ⟹ (A = U) ∨ (∃ B .(B <: A))"
⟨*proof*⟩

lemma *obj_inv''* : "e,L ⊢ Obj f U : A ⟹ (A = U) ∨ (∃ B . B ≠ A ∧
(B <: A) ∧ e,L ⊢ Obj f U : B)"
⟨*proof*⟩

lemma *obj_inv_end'* : "⟦e,L ⊢ Obj f U : U ⟧ ⟹ ¬ (∃ B . B ≠ U ∧ (
B <: U) ∧ e,L ⊢ Obj f U : B)"
⟨*proof*⟩

lemma *obj_inv_end''* : "⟦e,L ⊢ Obj f U : A ⟧ ⟹ (A ≠ U) ⟶ (∃ B .
B ≠ A ∧ (B <: A) ∧ e,L ⊢ Obj f U : B)"
⟨*proof*⟩

lemma *obj_inv_end'''* : "⟦e,L ⊢ Obj f U : A ⟧ ⟹ (A ≠ U) ⟶ ((
U <: A) ∧ e,L ⊢ Obj f U : U)"
```
```

⟨*proof*⟩

lemma *obj_inv_end_simp : "⟦e,L ⊢ Obj f U : A ⟧ ⟹ ((U <: A) ∧ e,L ⊢ Obj f U : U)"*
⟨*proof*⟩

An object's most special type is the annotation; any other type an object may have is a supertype thereof.

lemma *obj_inv_elim : "⟦e,L ⊢ Obj f U : U ⟧ ⟹ (dom f = do U ∧ (∀l ∈ (do U) . e<0:U,param (the U^l)>,L ⊢ (the (f l)) : fst (the (U^l))))"*
⟨*proof*⟩

A well-typed object has all methods described in its type annotation.

lemma *dom_lem: "e,L ⊢ Obj f (Object fun) : Object fun ⟹ dom f = dom fun"*
⟨*proof*⟩

Elimination rule for method invocation.

lemma *abs_typeE: "e,L ⊢ (Call (Obj f U) l b) : T ⟹ (⋀ V. ((e<0:U,param (the U^l)>),L ⊢ the (f l) : T) ⟹ P) ⟹ P"*
⟨*proof*⟩

Lifting preserves well-typedness

Lifting does not change the domain.

lemma *lift_option_dom [simp] : " ∀ b. dom (λl. lift_option (i) (b l)) = dom b"*
⟨*proof*⟩

lemma *lift_lift_option : "∀ l ∈ dom b . (lift (the (b l)) x) = the (lift_option x (b l))"*
⟨*proof*⟩

Lifting maintains typing.

lemma *lift_type [intro!]: " e,L ⊢ t : T ⟹ (⋀ U V. ∀ i ≤ env_length e . (e⟨i:U,V⟩,L ⊢ lift t i : T))"*
⟨*proof*⟩

lemma *select_suc [simp] : "⟦i < length list ⟧ ⟹((a # list) ! (Suc i)) = list!i"*
⟨*proof*⟩

Substitution preserves Well-Typedness

Substitution does not change domains.

lemma *subst_option_dom [simp] : " dom (λl. subst_option i x y (b l)) = (dom b)"*
⟨*proof*⟩

lemma *subst_option : "∀ l ∈ dom b . ((the (b l))[y,z/x]) = the (subst_option x y z(b l))"* ⟨*proof*⟩

Substitution maintains typing (under some circumstances).

lemma *subst_lemma:*
" e,L ⊢ t : T ⟹ (⋀e' i U u v V. i ≤ env_length e' ∧ e',L ⊢ u : U ∧ e',L ⊢ v : V ⟹ e = e'⟨i:U,V⟩ ⟹ e',L ⊢ t[u,v/i] : T)"
⟨*proof*⟩

Subject Reduction

First part of Type Soundness.

lemma *type_dom [simp] : "⟦env,L ⊢ (Obj a A) : A⟧ ⟹ dom a = do A"* ⟨*proof*⟩

Invocation yields the expected type-

lemma *select_preserve_type [simp] : " !! l1 l2 f env t n. ⟦env,L ⊢ Obj f (Object t) : Object t; env⟨0:(Object t), param (the (t l2))⟩,L ⊢ n : (fst (the (t l2))); l1 ∈ dom t; l2 ∈ dom t ⟧*
⟹ env⟨0:(Object t),param (the (t l1))⟩,L ⊢ (the ((f(l2 |-> n)) l1)) : (fst (the (t l1)))"

⟨*proof*⟩

lemma *dB_induct3:*
"[| !!n. P (Var (n));!!f T. Q f ==> P (Obj f T);
!!d1 d2 l.[| P d1; P d2 |] ==> P (Call d1 l d2);
!!d1 l d2. [| P d1; P d2 |] ==> P (Upd d1 l d2);
!!x. Q empty;
!!d1 f l. [| l ~: dom f; P d1; Q f |] ==> Q (f(l |-> d1));
!!d1 i. P d1 ==> P (dB.Asp_Label i d1) |]
==> P db & Q f"

⟨*proof*⟩

A well-typed term is also well-typed in any bigger environment.

lemma *empty_env' : " ⟦env1,L ⊢ t : A⟧ ⟹ ((env1+x),L ⊢ t : A)"*
⟨*proof*⟩

lemma *empty_env'' : " !! x env . ⟦env,L ⊢ t : A⟧ ⟹ ((env+x),L ⊢ t : A)"* ⟨*proof*⟩

Typed in the Empty Environment implies typed in any Environment (no free variables).

lemma *empty_env : " ⟦ Env [],L ⊢ t : A ⟧ ⟹ (env,L ⊢ t : A)"*
⟨*proof*⟩

lemma *empty_env_shift : "⟦((Env [])<0:X,Y>),L ⊢ t : A ⟧*
⟹ ((env<0:X,Y>),L ⊢ t : A)"
⟨*proof*⟩

Variables can have super/subtypes of the expected types.

lemma *var_subsumption' :*
" e,L ⊢ t : T
⟹ (⋀ e' A B C D x. x ≤ env_length e' ∧ e = e'⟨x:B,D⟩
∧ A <: B ∧ C <: D ⟹ e'⟨x:A,C⟩,L ⊢ t : T)"
⟨*proof*⟩

lemma *var_subsumption: "⟦e⟨x:B,D⟩,L ⊢ t : T; x ≤ env_length e; A <: B; C <: D⟧ ⟹ e⟨x:A,C⟩,L ⊢ t : T"*
⟨*proof*⟩

Main Lemma

lemma *subject_reduction: "e,L ⊢ t : T ⟹ (⋀t'. t -> t' ⟹ e,L ⊢ t' : T)"*
⟨*proof*⟩

theorem *subject_reduction': "t →β* t' ⟹ e,L ⊢ t : T ⟹ e,L ⊢ t' : T"*
⟨*proof*⟩

Properties to establish Progress

lemma *type_members_equal : "⟦do A = do B; ∀i. A^i = B^i⟧ ⟹ A = B"*
⟨*proof*⟩

If it's typed in the empty environment, it can't be a variable

lemma *not_var : " ⟦Env [],L ⊢ a : A ⟧ ⟹ ∀ x. a ≠ Var x"*
⟨*proof*⟩

A well typed invocation calls an existing method.

lemma *Call_label_range: "⟦(Env []),L ⊢ Call (Obj c T) l b : A ⟧ ⟹ l ∈ dom c"*
⟨*proof*⟩

lemma *Call_subterm_type: "⟦Env [],L ⊢ Call t l b: T0 ⟧ ⟹ (∃ T. Env [],L ⊢ t : T) ∧ (∃ T. Env [],L ⊢ b : T) "*

⟨*proof*⟩

lemma *Upd_label_range: "!!x.⟦Env [],L ⊢ Upd (Obj c T) l x : A ⟧ ⟹l ∈ dom c"*
⟨*proof*⟩

lemma *Upd_subterm_type: "⟦ Env [],L ⊢ Upd t l x : T0 ⟧ ⟹ (∃ T. Env [],L ⊢ t : T) "* ⟨*proof*⟩

lemma *no_var :" ⟦ ∃ T . Env [],L ⊢ Var x :T ⟧ ⟹ false"*
⟨*proof*⟩

Delabelling

A function to remove all aspect labels after weaving.

```
primrec
  delabel :: "dB ⇒ dB"
and
  delabel_option :: "[ dB option ] => dB option"
where
   "delabel  (Var n) = Var n"
   |"delabel (l ⟨ t ⟩ )  = delabel t"
   |"delabel (Call s l b)  = Call (delabel s) l (delabel b)"
   |"delabel (Upd s l t) = Upd (delabel s) l (delabel t)"
   |"delabel (Obj f B) = Obj (λ  l. (delabel_option (f l))) B"
   |"delabel_option None = None"
   |"delabel_option (Some t) = Some (delabel t)"
```

A function that establishes whether a term has labels.

```
primrec
  nolabel :: "dB ⇒ bool "
and
  nolabel_option :: "[ dB option ] => bool"
where
   "nolabel  (Var n) = True"
   |"nolabel (l ⟨ t ⟩ )  = False"
   |"nolabel (Call s l b )  = ((nolabel s) ∧ (nolabel b))"
   |"nolabel (Upd s l t) = ((nolabel s) ∧ (nolabel t))"
   |"nolabel (Obj f B) = ( ∀ l . nolabel_option  (f l))"
   |"nolabel_option None = True"
   |"nolabel_option (Some t) = nolabel t"
```

lemma *delabel_option_dom [simp] : " dom (λl. delabel_option (b l)) = (dom b)"*
⟨*proof*⟩

Delabelling is idempotent.

lemma *delabel_idempotent' :"(delabel t = delabel (delabel t)) & (! T. (delabel (Obj f T) = delabel (delabel (Obj f T))))"*
⟨*proof*⟩

lemma *delabel_idempotent : "(delabel t = delabel (delabel t))"* ⟨*proof*⟩

lemma *delabel_not_a_label': "(! i x . (delabel t ≠ i ⟨x⟩)) & (! x i T . (delabel (Obj f T) ≠ i ⟨x⟩)) "*
⟨*proof*⟩

lemma *delabel_not_a_label'' : "(! i x . (delabel t ≠ i ⟨x⟩)) "* ⟨*proof*⟩

lemma *delabel_not_a_label : "~ (∃ t'. i ⟨ d1 ⟩ = delabel t')"*
⟨*proof*⟩

Delabel is transparent w.r.t. typing.

lemma *delabel_preserves : "e,L ⊢ t : A ⟹ e,L ⊢ delabel t : A"*
⟨*proof*⟩

Progress

Final Type Soundness Lemma

lemma *pre_progress : "(!A. (∃ t' . t= delabel t') & Env [],L ⊢ (t) : A ⟶ ¬(∃ c T . ((t) = Obj c T)) ⟶ (∃ b . ((t) →β b))*
& (!A. Env[],L ⊢ Obj f A : A ⟶ ¬(∃ c T. (Obj f A = Obj c T)) ⟶ (∃ b . (Obj f A →β b)))"

⟨*proof*⟩

theorem *progress : " ⟦ (∃ t' . t= delabel t') ; Env [],L ⊢ t : A; ¬(∃ c A. (t = Obj c A))⟧ ⟹ (∃ b . (t →β b)) "*
⟨*proof*⟩

theorem *progress'' : " ⟦ Env [],L ⊢ (delabel t): A; ¬(∃ c A. ((delabel t) = Obj c A))⟧ ⟹ (∃ b . ((delabel t) →β b)) "*
⟨*proof*⟩

end

A.5 Aspect Orientation

theory *Weaving* **imports** *"../coreCalculus/Sigma"* **begin**

Aspects

An Aspect consists of a list of labels and a naked method body. The self parameter will be bound to the base term, in the vein of Clifton/Leavens, using Ligatti et al style Labels.

datatype *aspect = ASP "nat list" dB ("_._")*

primrec
pc :: "aspect ⇒ nat list"
where
"pc (ASP poc advi) = poc"

primrec
adv :: "aspect ⇒ dB"
where
"adv (ASP poc advi) = advi"

Weaving function

A simple notion of weaving - the aspect code is woven into the existing term, creating a well-typed term, iff the base term was well typed. We see this approach as static weaving.

primrec
weave :: "[dB, aspect] => dB"
and
weave_option :: "[dB option, aspect] => dB option"
where
WeaveVar: "weave (Var n) a = Var n"
|WeaveLab: "weave (l ⟨ t ⟩) a = (if (l mem (pc a)) then (l ⟨ (adv a)[(weave t a),(Obj empty (Object empty))/0] ⟩) else (l ⟨ (weave t a) ⟩))"
|WeaveCall: "weave (Call s l t) a = Call (weave s a) l (weave t a)"
|WeaveUpd: "weave (Upd s l t) a = Upd (weave s a) l (weave t a)"
|WeaveObj: "weave (Obj f B) a = Obj (λ l. (weave_option (f l) a)) B"
|WeaveNone: "weave_option None a = None"
|WeaveSome: "weave_option (Some t) a = Some (weave t a)"

constdefs *Weave :: "[dB, aspect list] => dB" "Weave t l == foldl weave t l"*

Basic lemmas about weaving

lemma *dom_weaveoption_lem: "dom (λ l . weave_option (f l) A) = dom f"*
⟨*proof*⟩

lemma *weave_option_lem : "(l::Label) ∈ dom (b::(Label -˜> dB)) ⟹ (the (weave_option (b l) a)) = (weave (the (b l)) a)"*
⟨*proof*⟩

lemma *empty_pc_weaving' : "((pc asp) = [] ⟶ ((weave t asp) = t)) ∧ (! T. (pc asp) = [] ⟶ (weave (Obj f T) asp = (Obj f T)))"*

⟨*proof*⟩

lemma *empty_pc_weaving : "⟦ (pc asp) = [] ⟧ ⟹ ((weave t asp) = t)"*
⟨*proof*⟩
end

theory *Compositionality* **imports** *"../coreCalculus/Confluence" Weaving*
begin

Aspect Compositionality

consts *FVcalc :: "dB ⇒ nat ⇒ dB set"*

primrec *FVtestvar :: "Variable ⇒ nat ⇒ bool"*
where
FVSelf: "FVtestvar (Self x) n = (x < n)"
|FVParam: "FVtestvar (Param x) n = (x < n)"

primrec *FVtest :: "dB ⇒ nat ⇒ bool"*
and
optionFVtest :: "dB option ⇒ nat ⇒ bool"
where
FVVar: "FVtest (Var i) n = FVtestvar i n"
|FVActive: "FVtest (Asp_Label m a) n = FVtest a n"
|FVCall: "FVtest (Call a l b) n = (((FVtest a) n) ∧ ((FVtest b) n))"
|FVUpd: "FVtest (Upd a l b) n = ((FVtest a n) & (FVtest b (n+1)))"
|FVObj: "FVtest (Obj f T) n = fold (λx y. y ∧ (optionFVtest (f x) (n+1))) True (dom f) "
|FVoption: "optionFVtest None n = True"

```
  |FVoption':  "optionFVtest (Some a) n = (FVtest a n)"
```

constdefs *noFV :: "dB ⇒ bool" "noFV t == FVtest t 0"*

constdefs *justoneFV :: "dB ⇒ bool" "justoneFV t == FVtest t 1"*

lemma *dB_induct: "⟦⋀nat. P (Var nat); ⋀fun type. (⋀x. Q (fun x)) ⟹ P (Obj fun type);*
⋀dB1 Label dB2. ⟦P dB1; P dB2⟧ ⟹ P (Call dB1 Label dB2);
⋀dB1 Label dB2. ⟦P dB1; P dB2⟧ ⟹ P (Upd dB1 Label dB2); ⋀nat dB. P dB ⟹ P nat⟨dB⟩;
Q None; ⋀dB. P dB ⟹ Q (Some dB)⟧
⟹ P dB ∧ Q option"
⟨*proof*⟩

lemma *fold_implies':" ⟦finite B; Finite_Set.fold_graph (λ x y. y ∧ P x) True B x⟧ ⟹(∀ x ∈ B. P x) ⟶ x"*
⟨*proof*⟩

lemma *fold_implies:" ⟦finite B; (∀ x ∈ B. P x); Finite_Set.fold_graph (λ x y. y ∧ P x) True B x⟧ ⟹ x"*
⟨*proof*⟩

lemma *fold_lem: "⟦finite B; (∀ x ∈ B. P x)⟧ ⟹ fold (λ x y. y ∧ P x) True B "*
⟨*proof*⟩

interpretation *fun_left_comm "(λ x y. y ∧ P x)"*
⟨*proof*⟩
lemma *fold_lem_inv: "⟦finite B; fold (λ x y. y ∧ P x) True B; x: B ⟧ ⟹ P x"*
⟨*proof*⟩

lemma *fold_FVlem:*
"⟦finite (dom fun); fold (λx y . y ∧ optionFVtest (fun x) (Suc n)) True (dom fun); fun l = Some a⟧ ⟹ optionFVtest(fun l)(Suc n)"

⟨*proof*⟩

lemma *nFV_eq[rule_format] : "(!n. FVtest t n ⟶ lift t n = t)&(! n. FVtest (the s) n ⟶ lift_option n s = s)"*
⟨*proof*⟩

lemma *nFV_eq' : "FVtest t n ⟹ lift t n = t"*

⟨*proof*⟩

lemma *justoneFV_eq[rule_format] : "(justoneFV t ⟶ lift t 1 = t)"*
⟨*proof*⟩

lemma *nFVsubst [rule_format]: "(! m. FVtest t m ⟶ (! s v. ! n. m < n ⟶ t [s,v/n] = t))&*
(! m. FVtest (the t') m ⟶ (! s v. ! n. m < n ⟶ subst_option n s v t' = t'))"
⟨*proof*⟩

lemma *nFVsubst'' [rule_format]: "(! n s v. FVtest t n ⟶ t [s,v/n] = t)&*
(! n s v. FVtest (the t') n ⟶ subst_option n s v t' = t')"
⟨*proof*⟩

lemma *noFVsubst [rule_format]: "noFV t ⟶ t [s,v/ Suc n] = t"*
⟨*proof*⟩

lemma *noFVsubst_zero [rule_format]: "noFV t ⟶ t [s,v/0] = t"*
⟨*proof*⟩

lemma *noFVsubst''' [rule_format]: "⟦noFV t⟧ ⟹ t [s,v/n] = t"*
⟨*proof*⟩

lemma *justoneFVsubst_zero [rule_format]: "justoneFV t ⟶ t [s,v/Suc 0] = t"*
⟨*proof*⟩

lemma *justoneFVsubst_nozero [rule_format]: "(justoneFV t & n > 0 ⟶ t [s,v/ Suc n] = t)"*
⟨*proof*⟩

lemma *justoneFVsubst [rule_format]: "(justoneFV t ⟶ t [s,v/ Suc n] = t)"*
⟨*proof*⟩

lemma *FVtest_leq [rule_format]: "(! n. FVtest t n ⟶ (! m. n <= m ⟶ FVtest t m))&*

```
                                    (! n. optionFVtest t' n ⟶ (! m. n <=
m ⟶ optionFVtest t' m))"
```
⟨*proof*⟩

lemma *justoneFV_leq[rule_format] : "(justoneFV t ∧ n >= 0 ⟶ lift t (Suc n) = t)"*
⟨*proof*⟩

lemma *lift_subst_adv: "justoneFV a ⟹ (lift (a[s,v/0]) 0) = a [(lift s 0),(lift v 0)/0]"*
⟨*proof*⟩

lemma *lift_subst_n_adv: "⟦ justoneFV a; n >= 0 ⟧ ⟹ (lift (a[s,v/0]) n) = a [(lift s n),(lift v n)/0]"*
⟨*proof*⟩

lemma *weave_lift_adv [rule_format]:*
```
     "(∀ n. justoneFV (adv a) ⟶ weave (lift s n) a = lift (weave s
a) n) &
      (∀ n. justoneFV (adv a) ⟶ weave_option (lift_option n t) a =
lift_option n (weave_option t a))"
```

⟨*proof*⟩

lemma *weave_lift: "justoneFV (adv a) ⟹ weave (lift s 0) a = lift (weave s a) 0"*
⟨*proof*⟩

lemma *noFVlift_zero [rule_format]: "(noFV t ⟶ noFV(lift t 0))"*
⟨*proof*⟩

lemma *comp_weave_subst [rule_format]:*
```
   "(! a n s v. justoneFV (adv a)   ⟶ (weave(t[s,v/n]) a) = ((weave
t a)[(weave s a),(weave v a)/n]))
  & (! a n s v. justoneFV (adv a)   ⟶ (weave_option (subst_option n
s v t') a) =
                          (subst_option n (weave s a)(weave v a)(weave_option
t' a)))"
```

⟨*proof*⟩

lemma *comp_weave_subst':*

```
   "justoneFV (adv a) ⟹ (weave(t[s,v/n]) a) = ((weave t a)[(weave s
a),(weave v a)/n])"
```
⟨*proof*⟩

lemma *comp_weave_par [rule_format]:*
```
   "⟦ justoneFV (adv a);   t -> t' ⟧ ⟹ (weave t a) => (weave t' a)"
```

⟨*proof*⟩

lemma *comp_weave [rule_format]:*
```
   "⟦ justoneFV (adv a); t -> t'⟧ ⟹ (weave t a) ->> (weave t' a)"
```
⟨*proof*⟩

end

theory *TypeSafety* **imports** *Weaving Compositionality "../types/TypedSigma"*
begin

Aspect Safety

Aspect well-formedness

An aspect is well formed, if applying it to a base term of a given type preserves that type in the empty environment. The base term types are defined by a list of labels. This is regardless of the actual base-term.

constdefs *void :: type*
```
"void == Object empty"
```

lemma *empty_void: "enva,La⊢ Obj empty (Object empty) : void"*
⟨*proof*⟩

constdefs *wf_adv :: "[type list, aspect] => bool"*
```
          "wf_adv  L a == ∀ l . (l mem (pc a)) ⟶ ((Env []<0:(L!l),void>),L
⊢ (adv a): (L!l))"
```

An aspect is compatible to a given base term, if the aspect is well formed and the base application is well-typed using the same label list.

constdefs *wf_at :: "[type list, type, dB, aspect] => bool"*
```
    "wf_at L T t a ==  (Env [], L ⊢ t: T) ∧ (wf_adv  L a)"
```

Using many aspects.

constdefs *wf :: "[type list, type, dB, aspect list] => bool"*
```
    "wf L T t A ==  (Env [], L ⊢ t: T) ∧ (∀  a ∈ (set A). wf_at L T
t a)"
```

primrec
wf_test :: "[type list, nat list, dB] => bool"
where
"wf_test L [] ad = True"
|"wf_test L (a#p) ad = ((((Env [])<0:(L!a),void>),L ⊢ (ad): (L!a)) ∧ (wf_test L p ad))"

Basic lemmas about weaving

lemma *fewer_aspects : "wf L T t (a # A) ⟹ wf L T t A"*
⟨*proof*⟩

lemma *dom_weaveoption_lem: "dom (λ l . weave_option (f l) A) = dom f"*
⟨*proof*⟩

lemma *rev_wf: " wf L t T A ⟹ wf L t T (rev A)"*
⟨*proof*⟩

lemma *weave_option_lem : "(l::Label) ∈ dom (b::(Label -~> dB)) ⟹ (the (weave_option (b l) a)) = (weave (the (b l)) a)"*
⟨*proof*⟩

Weaving of well-formed aspects yields safe terms

lemma *weaving_preservation': "!! env L A . ⟦ wf_adv L A; Env [], L ⊢ t : T ⟧ ⟹ wf_adv L A ⟶ Env [], L ⊢ weave t A : T "*
⟨*proof*⟩
thm *T_lab*

⟨*proof*⟩

lemma *weaving_preservation'': "!! env L A . ⟦ wf_adv L A; Env [], L ⊢ t : T ⟧ ⟹ Env [], L ⊢ weave t A : T "*
⟨*proof*⟩

lemma *weaving_preservation_wf: "!! env L A . ⟦ wf L T t (a#A) ⟧ ⟹ wf L T (weave t a) A "*
⟨*proof*⟩

theorem *weaving_preservation: "⟦ wf L T t A ⟧ ⟹ Env [], L ⊢ Weave t A : T"*

⟨*proof*⟩

theorem *aspect_preservation_delabel : "⟦ wf L T t A; delabel(Weave t A) -> t' ⟧ ⟹ e, L ⊢ t': T"*
⟨*proof*⟩

theorem *aspect_preservation: "⟦ wf L T t A; Weave t A -> t' ⟧ ⟹ e, L ⊢ t': T"*
⟨*proof*⟩

theorem *aspect_progress: "⟦ wf L T t A; ~(∃ c T . delabel (Weave t A) = Obj c T) ⟧*
⟹ (∃ t'. delabel (Weave t A) -> t')"
⟨*proof*⟩
thm *progress*
⟨*proof*⟩

constdefs *Weave' :: "[dB, aspect list] => dB" "Weave' t l == delabel (Weave t l)"*

theorem *aspect_preservation': "⟦ wf L T t A; Weave' t A -> t' ⟧ ⟹ e, L ⊢ t': T"*
⟨*proof*⟩

theorem *aspect_progress': "⟦ wf L T t A; ~(∃ c T . (Weave' t A) = Obj c T) ⟧*
⟹ (∃ t'. (Weave' t A) -> t')"
⟨*proof*⟩

Properties of typed Aspects

Well–formed aspects are typed. Typed terms have no free variables and are thus compositional

lemma *env_len_FV' : " e,L ⊢ t : T ⟹ (!! i. env_length e <= i ⟹ FVtest t i)"*
⟨*proof*⟩
thm *conjI*
⟨*proof*⟩

theorem *env_len_FV : " ⟦e,L ⊢ t : T⟧ ⟹ FVtest t (env_length e)"* ⟨*proof*⟩

lemma *env_len_no_FV : "⟦Env [],L ⊢ t : T ⟧ ⟹ noFV t"*
⟨*proof*⟩

lemma *env_len_one_FV : "⟦Env []⟨0:A,void⟩,L ⊢ t : T ⟧ ⟹ justoneFV t"*

⟨*proof*⟩

theorem *WFAsp_typed : "⟦wf_adv L a⟧ ⟹ ((pc a) = []) ∨ (∃ A .((Env []⟨0:A,void⟩,L ⊢ (adv a) : A)))"*
⟨*proof*⟩

theorem *typed_or_emty_comp : "⟦((pc a) = []) ∨ (∃ A .((Env []⟨0:A,void⟩,L ⊢ (adv a) : A))); t -> t'⟧ ⟹ (weave t a) ->> (weave t' a)"*
⟨*proof*⟩

theorem *WFAsp_comp : "⟦wf_adv L a ; t -> t'⟧ ⟹ (weave t a) ->> (weave t' a)"*
⟨*proof*⟩

end

Bibliography

M. Abadi and L. Cardelli. A theory of primitive objects. In *Proceedings Of Theoretical Aspect Of Computer Software*, volume 789 of *LNCS*, pages 296–320. Springer-Verlag, 1994.

M. Abadi and L. Cardelli. An imperative object calculus. In *TAPSOFT '95: Theory and Practice of Software Development*, volume 915 of *LNCS*, 1995.

M. Abadi and L. Cardelli. *A Theory of Objects (Monographs in Computer Science)*. Springer, 1998.

J. Aldrich. Open modules: Modular reasoning about advice. In *European Conference on Object Oriented Programming*, volume 3586 of *LNCS*. Springer, 2005.

AspectJ. The aspectj project, 2009. URL `http://www.eclipse.org/aspectj/`.

P. Avgustinov, E. Hajiyev, N. Ongkingco, O. de Moor, D. Sereni, J. Tibble, and M. Verbaere. Semantics of static pointcuts in aspectj. In *Proceedings of the 2007 POPL Conference*, volume 42, pages 11–23, New York, NY, USA, 2007. ACM.

B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *ACM Symposium on Principals of Programming Languages (POPL) 2008*, volume 43, pages 3–15, New York, NY, USA, 2008. ACM.

B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich1, and S. Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. http://www.cis.upenn.edu/ plclub/wiki-static/poplmark.pdf, 2007.

H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, 1984.

H. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

S. Berghofer and C. Urban. A head-to-head comparison of de bruijn indices and names. *ENTCS*, 174(5):53–67, 2007.

S. Berghofer and M. Wenzel. Inductive datatypes in HOL - lessons learned in formal-logic engineering. In *Theorem Proving in Higher Order Logics, TPHOLs'99*, volume 1690 of *LNCS*. Springer, 1999.

S. Berghofer, L. Bulwahn, and F. Haftmann. Turning inductive into equational specifications. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *LNCS*, Berlin, Heidelberg, 2009. Springer.

Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. V. Tassel. Experience with embedding hardware description languages in hol. In *Theorem Provers in Circuit Design: Proceedings of the IFIP TC10/WG 10.2 International Conference*, pages 129–156. North-Holland, 1992.

N. G. D. Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

N. G. D. Bruijn. Checking mathematics with computer assistance. *Notices of the American Mathematical Society*, 38:8–15, 1991.

A. Charguéraud. The locally nameless representation. Thesis draft, 2009.

A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

A. Church. A formulation of the simple theory of types. *Journal for Symbolic Logic*, 5(2):56–68, 1940.

A. Ciaffaglione, L. Liquori, and M. Miculan. Reasoning on an imperative object-based calculus in higher order abstract syntax. In *MERLIN '03: Proceedings of the 2003 ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding*, pages 1–10, New York, NY, USA, 2003. ACM.

C. Clifton, G. T. Leavens, and M. Wand. Parameterized aspect calculus: A core calculus for the direct study of aspect-oriented languages. Technical Report 3-13, Dept. of Computer Science, Iowa State University, 2003.

C. Constantinides, T. Skotiniotisa, and M. Stoerzer. AOP considered harmful. In *EIWAS'04 European Interactive Workshop on Aspects in Software*, 2004.

W. Cook. A proposal for making eiffel type-safe. In *The Computer Journal*, pages 57–70. Cambridge University Press, 1989.

T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North Holland, 1958. Second edition, 1968.

D. S. Dantas and D. Walker. Harmless advice. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 383–396, New York, NY, USA, 2006. ACM.

B. De Fraine, M. Südholt, and V. Jonckers. Strongaspectj: flexible and safe pointcut/advice bindings. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, pages 60–71, New York, NY, USA, 2008. ACM.

R. E. Filman and D. P. Friedman. *Aspect-Oriented Programming is Quantification and Obliviousness*, pages 21–35. Addison-Wesley, 2005.

R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.

Flyspeck. The flyspeck project, 2009. URL `http://code.google.com/p/flyspeck/wiki/FlyspeckFactSheet`. Retrieved September 2nd 2009.

G. Frege. Begriffsschrift. *In Jean van Heijnoort,* From Frege to Gödel, *Harvard University Press, Cambridge, 1967*, N/A:N/A, 1879.

J.-Y. Girard. *Interpretation Fonctionelle et Elimination des Compures de l'Arithmetic d'Ordre Superieur*. PhD thesis, Universite Paris VII, 1972.

M. Gordon. *Proof, Language and Interaction: essays in honour of Robin Milner*, pages 169–186. MIT Press, 2000.

S. Gudmundson and G. Kiczales. Addressing practical software development issues in aspectj with a pointcut interface. In *In Advanced Separation of Concerns*, 2001.

C. h. L. Ong. Correspondence between operational and denotational semantics. In *Handbook of Logic in Computer Science*, pages 269–356. Oxford University Press, 1995.

F. Haftmann. Haskabelle homepage, 2009. URL `http://www.cl.cam.ac.uk/research/hvg/Isabelle/haskabelle.html`.

L. Hakobyan. Interpreter extraction for a functional calculus (working title). Diploma thesis, forthcoming, Freie Universität Berlin, 2010.

L. Henrio, F. Kammüller, B. Lutz, and H. Sudhof. A locally nameless theory of objects. In *SAFA 2010, to appear.*

L. Henrio, F. Kammüller, and H. Sudhof. Aspfun: A functional and distributed object calculus semantics, type-system, and formalization. Research Report 6353, INRIA, 11 2007. URL `http://hal.inria.fr/inria-00186963/fr/`.

S. Herrmann. A precise model for contextual roles: The programming language objectteams/java. *Applied Ontology*, 2(2):181–207, 2007.

J. R. Hindley. *Basic simple type theory.* Cambridge University Press, New York, NY, USA, 1997.

W. A. Howard. *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, chapter The formulae-as-types notion of construction, pages 479–490. Academic Press, Boston, 1969.

A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM SIGPLAN Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, Oct. 1999. Full version in ACM Transactions on Programming Languages and Systems (TOPLAS), 23(3), May 2001.

R. Jagadeesan, A. Jeffrey, and J. Riely. Typed parametric polymorphism for aspects. *Science of Computer Programming: Special Issue on Foundations of Aspect-Oriented Programming*, 63(3):267–296, 2006.

F. Kammüller. Modular reasoning in isabelle. PhD Thesis 470, Computer Laboratory, University of Cambridge, 1999.

F. Kammüller. Interactive theorem proving in software engineering. Habilitationsschrift, 2006.

F. Kammüller and H. Sudhof. A mechanized framework for aspects in Isabelle/HOL. In *2nd Informal ACM SIGPLAN Workshop on Mechanizing Metatheory*, 2007a.

F. Kammüller and H. Sudhof. A formalization of typed aspects for the ς-calculus in Isabelle/HOL. Technical Report 2007-20, Technische Universität Berlin, Fakultät IV, 2007b.

F. Kammüller and H. Sudhof. Composing safely – a type system for aspects. In *7th International Symposium on Software Composition, SC 2008*, volume 4954 of *LNCS*. Springer, 2008a.

F. Kammüller and H. Sudhof. *Autonomous Systems – Self-Organization, Management, and Control*, chapter Compositionality of Aspect Weaving, pages 87–95. Springer, 2008b.

F. Kammüller and H. Sudhof. Ascot homepage, 2009a. URL `http://swt.cs.tu-berlin.de/~flokam/ascot/index.html`.

F. Kammüller and H. Sudhof. A mechanized theory of aspects. In *Theorem Proving in Higher Order Logics, TPHOLs 2009, Emerging Trends*, 2009b. Proceedings as Technical Report TUM-I0916, Technische Universität München.

E. Katz and S. Katz. Modular verification of strongly invasive aspects: summary. In *FOAL 2009*, pages 7–12, 2009.

S. Katz. Aspect categories and classes of temporal properties. *Transactions on Aspect-Oriented Software Development I*, 3880:106–134, 2006.

G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ICSE*, pages 49–58, 2005.

G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. V. Lopes, C. Maeda, and A. Mendhekar. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, 1997.

G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

S. Kleene. Disjunction and existence under implication in elementary intuitionistic formalisms. *Journal of Symbolic Logic*, 27:11–18, 1962.

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.

T. Kleymann and D. Aspinall. Proof general, 2009. URL `http://proofgeneral.inf.ed.ac.uk/main`.

D. E. Knuth. backus normal form vs. backus naur form. *Commun. ACM*, 7 (12):735–736, 1964.

X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52:107–115, 2009.

X. Leroy and H. Grall. Coinductive big-step operational semantics. *Information and Computation.*, 207:284–304, 2009.

P. Letouzey. Extraction in coq: An overview. In *CiE*, pages 359–369, 2008.

J. Ligatti, D. Walker, and S. Zdancewic. A type-theoretic interpretation of pointcuts and advice. *Science of Computer Programming: Special Issue on Foundations of Aspect-Oriented Programming*, 63(3):240–266, 2006.

A. Lochbihler. Formalising finfuns - generating code for functions as data from Isabelle/HOL. In *Theorem Proving in Higher Order Logics 2009*, volume 5674 of *LNCS*. Springer, 2009.

B. Lutz. A locally nameless object calculus. Bachelor thesis, forthcoming, Technische Universität Berlin, 2010.

D. Mackenzie. What in the name of euclid is going on here? *Science*, 307: 65–80, 2005.

P. Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory.* Bibliopolis, Naples, 1984.

R. Milner. Logic for computable functions. Technical Report STAN-CS-72-288, Stanford University, 1972.

P. Naur and B. Randell. *Software Engineering: Report On A Conference Sponsored By The Nato Science Committee.* The Nato Science Committee, 1968.

T. Nipkow. More Church-Rosser proofs (in Isabelle/HOL). In M. McRobbie and J. Slaney, editors, *Automated Deduction — CADE-13*, volume 1104 of *LNCS*, pages 733–747. Springer, 1996.

T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, Berlin, 2002.

L. C. Paulson. Isabelle: The next seven hundred theorem provers. In *9th International Conference on Automated Deduction, CADE*, volume 310 of *Lecture Notes in Computer Science*. Springer, 1988.

L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

L. C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, New York, NY, USA, 1990. ISBN 0521395607.

F. Pfenning and C. Elliott. Higher-order abstract syntax. In *PLDI '88: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 1988.

B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

G. D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:3–15, 2004.

J. C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, Berlin, 1974. Springer-Verlag.

J. C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

J. B. Rosser. Highlights of the history of the lambda-calculus. In *LFP '82: Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 216–225, New York, NY, USA, 1982. ACM.

B. Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30(3):222–262, 1908. In Jean van Heijnoort, *From Frege to Gödel*, Harvard University Press, Cambridge, USA, 1967.

A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21:2003, 2003.

D. Scott. A type-theoretical alternative to iswim, cuch, owhy. *Theoretical Computer Science*, 121:411–440, 1993. original from 1969.

D. Sokenou, K. Mehner, S. Herrmann, and H. Sudhof. Patterns for re-usable aspects in object teams. In *NODe/GSEM*, pages 65–80, 2006.

F. Steimann. The paradoxical success of aspect-oriented programming. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications*, pages 481–497, New York, NY, USA, 2006. ACM Press.

F. Steimann, T. Pawlitzki, S. Apel, and C. Kästner. Types and modularity for implicit invocation with implicit announcement. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Dec. 2009. accepted for publication.

H. Sudhof. Vergleichende Fallstudie über Techniken für Wiederverwendbare Aspekte. Diplomarbeit, Technische Universität Berlin, 2006.

R. Thiemann and C. Sternagel. Certification of termination proofs using ceta. In *Proceedings TPHOLs '09*, LNCS 5674, pages 452–468. Springer, 2009.

C. Urban. Nominal techniques in Isabelle/HOL. *J. Autom. Reasoning*, 40 (4):327–356, 2008.

C. Urban. Nominal methods group, 2009. URL `http://www4.in.tum.de/urbanc/Nominal/`.

J. van Heijenoort. *From Frege to Gödel.* Harvard University Press, Cambridge, USA, 1967.

A. N. Whitehead and B. Russell. *Principia Mathematica, Volume I.* Cambridge University Press, 1910.

A. N. Whitehead and B. Russell. *Principia Mathematica, Volume II.* Cambridge University Press, 1912.

A. N. Whitehead and B. Russell. *Principia Mathematica, Volume III.* Cambridge University Press, 1913.

F. Wiedijk. *The Seventeen Provers of the World*, volume 3600 of *LNCS.* Springer, 2006.

G. Winskel. *The Formal Semantics of Programming Languages.* MIT Press, 1993.

A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1992.

Index