Technische Universität Berlin

Forschungsberichte
der Fakultät IV – Elektrotechnik und Informatik

A Formalization of Typed Aspects for the ς-calculus In Isabelle/HOL

Florian Kammüller and Henry Sudhof

Bericht-Nr. 2007 – 20

ISSN 1436-9915

A Formalization of Typed Aspects for the ς-calculus in Isabelle/HOL

Florian Kammüller and Henry Sudhof

Technische Universität Berlin
Insititut für Softwaretechnik und Theoretische Informatik

Abstract

Abstract. In this paper we present an approach towards safe software composition based on aspect-orientation. Aspects enable the systematic addition of code into existing programs but often they also introduce errors. In order to provide safe aspects for software composition we address the verification of the aspect-oriented language paradigm. We construct a basic calculus for aspects with types and prove formally type safety. More precisely, this paper presents the following contributions (a) a fully formalized type system for the Theory of Objects including the proof of type safety, (b) a theory of aspects based on the Theory of Objects including a type system for aspects, and (c) the definition of a notion of type safety for aspects including its proof. The entire theory and proofs are carried out in the theorem prover Isabelle/HOL.

Table of Contents

1 Introduction

Aspect-orientation has enjoyed major attention for years and is supported by many major programming languages. There are, however, serious problems in the current implementations of aspect-oriented languages. In [11] we show how the lack of typing produces unforeseen runtime-errors. Jagadeesan et al. have more recently shown [9] that there are even contradictions in other seemingly simple situations. The described problem arises when using conform redefinition of functions in the base code, which causes a covariance issue at runtime.

But, even without considering inheritance, crashes can be produced. For instance, the predominant aspect-oriented language AspectJ still relies on partially untyped expressions, resulting in runtime failure. The program depicted in Figure 1 will compile without issue, but crash with a runtime error. This is obviously

```
public class Test
{
  public Test test()
  {
    return this;
  }
}
```

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

Fig. 1. This code compiles using the current AspectJ compiler. It terminates with a `ClassCastException` whenever `test` is called

conflicting with the expected behaviour of well-typed programs. In fact, here, aspects are not typed at all. This essentially removes the ability to view aspects as compositional modules, as the aspect breaks the base code without any static hint that it might do so.

Automated formal analysis with proof assistants provides a strong support for the analysis of safety properties of programming languages [10]. Our approach to support the verification of adaptive systems consists of providing a fully formalized basis for aspect-oriented programming in Isabelle/HOL [8]. We construct a core calculus of objects and aspects with types as an instance of the generic theorem prover Isabelle/HOL. The resulting framework serves to experiment with language features – like weaving functionality and pointcut selectors – and properties – like type safety and compositionality. At the same time, these experiments are on a firm basis. The results have mathematical precision and are mechanically verified. Moreover, we try to keep the formal model of the aspect calculus as constructive as possible. Thereby, we can extract executable protoypes for evaluators and type checkers from the Isabelle/HOL framework.

The basic idea of our calculus of aspects is similar to the theory of aspects [12] but we start from the Theory of Objects, unlike the former that is based on the λ-calculus. These models of aspects simply introduce labels in the base program. The labels represent so-called join-points, i.e. points at which advice might be woven in. [1] Given these labels, we can quite naturally define weaving. The idea is that advice is given as a function f that can be applied to a labelled term $l\langle t\rangle$, replacing the original term t bf $f(t)$. So, given an aspect as a pair $\langle L.f\rangle$ of pointcuts L and an advice f that shall be applied at all points specified by L, weaving can be simply constructed using function application, as illustrated in the following example, where weaving is represented as an infix downwards arrow $\Downarrow$ and functions and application using λ-calculus.

$$\langle L.\lambda\ x.\ e\rangle \Downarrow (v_1 + l_1\langle v_2\rangle) \xrightarrow{l_1 \in L} v_1 + e[v_2/x]$$

Moreover, we can now attach types to join-points by typing labels. Then, a failure like the one illustrated in Figure 1 would be detected at compile-time.

A major difficulty for the definition of a simple and precise calculus for aspect-orientation is *obliviousness* — one of the major criteria of aspect-oriented programming languages according to the widely accepted definition of Filman and Friedman [6]. Obliviousness means that a programmer can adapt a base program by aspects while being oblivious of the exact details of this base program. This serves to guarantee maximal freedom and flexibility of adaptation. At first sight, our concept of placing labels from start into a base program seems to clash with this idea. There are several answers to this. In our view, even though complete obliviousness might seem an appealing idea, it cannot be achieved. At least, the programmer has to be aware that there are points in a program where it is already syntactically impossible to add an advice, for example, in the middle of a keyword. Conceptually, we consider our aspect labels as the set of all syntactically possible join-points of a program. That is, all possible points where an advice might be woven into the base program are marked by a label. Using this global assumption, we lose no generality. Ligatti et al. [12] take a different line of argument to justify labels. They construct a core calculus that serves as the target code for a high-level aspect-oriented language. This high-level language is oblivious while the core calculus is not. However, type preserving compilation between the two yields property preservation. De Moore et al. produce yet another justification of the label concept [5] by showing through practical solution that pointcut descriptors may be statically resolved into labels. Hence, they practically show that labels do not interfer with obliviousness.

The remainder of the paper is organized as follows. We begin in Section 3 with a short presentation of the Theory of Objects and its formalization in Isabelle/HOL [16]. Section 4 is dedicated to the type system and proof of type safety for the ς-calculus. In Section 5 we introduce our extension of this base calculus to a calculus for aspects introducing weaving functionality and a type system for aspects. We present the definition and proof of type safety for aspects. Finally, we conclude with a comparison to other approaches and an outlook to future work in Section 6. Before we start delving into the technical presentation

[1] Representing pointcuts as sets of labels corresponds to the intuition that mathematically a predicate is equivalent to the set of all elements fulfilling that predicate. Thus the pointcut-selector predicate may as well be denoted by the set of all points that fulfill the predicate.

we use Section 2 to provide a proper introduction to the relevant features of the theorem prover Isabelle/HOL, a brief introduction to the ς-calculus, and some basic techniques we had to provide for our aspect theory.

2 Preliminaries

2.1 Isabelle/HOL

Isabelle [13] is an interactive ML-based theorem prover. It was initially developed by Lawrence Paulson at the University of Cambridge and is today maintained there and at the TU Munich. Unlike many other interactive provers, Isabelle was written to serve as a framework for various logics, so-called object-logics. Today, mostly the object-logic for Higher-Order-Logic (HOL) and – on a smaller scale – the one for Zermelo-Fraenkel set theory are in widespread use. Isabelle has a meta-logic serving as a deductive framework for the embedded object-logics. This meta-logic is itself a fragment of HOL solely consisting of the universal quantifier and the implication. Isabelle features a powerful simplifier, and automated proof strategies; moreover, it is supported by the generic ProofGeneral user interface. For this paper, Isabelle/HOL [16] was used, e.g. Isabelle in its instantiation to HOL. In Isabelle/HOL automatic code generation is possible for constructive parts of a formalization, like datatypes and inductive definitions (see below), but also for constructive proofs.

The following meta-logical formula is an example illustrating the universal quantification with $\bigwedge$, higher order variables `P` and `Q`, and implication $\Longrightarrow$ of Isabelle's meta-logic (the square brackets $[\![]\!]$ act as a pseudo-conjunction).

$$\bigwedge \mathtt{P\ Q\ x.}\ [\![\ \mathtt{P\ x;\ Q\ x}\ ]\!] \Longrightarrow \mathtt{P\ x}$$

The embedding of object-logics, like HOL, adds additional types, constants, and definitions gathered in a so-called *theory*. This extension of the meta-logic is performed according to a principle of *conservative* extension: new types and related constructors are defined on existing types; non-emtpiness of the new types must be proved; properties of new types are derived from their definition. Thereby, conservative extension guarantees soundness. Type definition for a restricted class of inductive types is more specifically supported by the *datatype* package in Isabelle/HOL. This feature resembles much an ML-style datatype definition. It is advisable to use this construction principle whenever possible for one's object-logic's specification because induction principles, distinction and exhaustion properties come along automatically with a structure defined as a datatype. In addition, functions over a datatype may be defined using primitive recursion which helps automated simplification in proofs and code generation from specifications. We will use the datatype feature, for example, to define the type of ς-terms in Section 3.1.

Isabelle/HOL features a simple concept of axiomatic type classes similar to the concept of Haskell. For the class of sets – corresponding one-to-one to sets in Isabelle/HOL – an additional inductive definition package enables the definition of a minimal set of elements closed under given inductive rules. We will use axiomatic type classes in Section 2.3 and inductive definition for the definitions of the type systems in Sections 4 and 5.

A very generic parser enables application-specific definition of concrete syntax (so called mixfix syntax) making Isabelle formulae and proofs almost identical to pen-and-paper formalizations. We will point out the use of mixfix syntax in our formalization. In general, any Isabelle/HOL specific syntax that we will be using throughout the paper is going to be explained when we use it.

2.2 The ς-calculus

In a *Theory of Objects*[1] Abadi and Cardelli developed the ς-family of calculi to formally study object-orientation. These calculi are widely accepted as conceptual equivalents of the λ-calculus for objects, since the objects can be directly used as a basic construct without having to be simulated through λ-expressions.

In the ς-calculi, an object is defined as a set of labelled methods. Each method is a ς-term in its own right and has a parameter *self*, in which the enclosing object is contained. There are three flavors of primitives from which to build such terms: *object definitions*, *method invocation* and *field update*, which are presented in Figure 2. Methods not using the self parameter are considered to be *fields*.

Let $o \equiv [li = \varsigma(x_i)bi^{i \in 1..n}]$ (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
$o.l_j \Leftarrow \varsigma(y)b \rightarrow [l_j = \varsigma(y)b, l_i = \varsigma(xi)b_i^{i \in (1..n)-j}](j \in 1..n)$ update / override

Fig. 2. The primitive semantics of the ς-calculus as introduced in [2]

2.3 Finite Maps for Isabelle/HOL

For the definition of ς-terms it is necessary to first introduce a generic type of finite maps for the representation of objects. Defining a type of finite maps is a simple enough exercise, but defining it in a generic way is quite tricky. Datatypes and primitive recursive function definitions cannot be defined over arbitrary types. If the defining predicate of a type is not inductive, such a type cannot be used in a recursive datatype, because the recursive type would not be inductive. As we need to use finite maps exactly for the recursive case of objects the only possibility is to construct finite maps in such a way that they can be used inside a datatype definition. However, finite maps cannot be defined as a datatype themselves because they should be functions and unordered. Such constraints cannot be expressed as provisos in the constructors of a datatype. One possible way out of this dilemma is to use axiomatic type classes to define finite maps which are inductive. We define a class `finite` of finite types as a subclass of the standard class `type` in Isabelle/HOL. Finite types will be used as domain types for finite maps. The proposition `finite_set` is the axiom that characterizes all types contained in the new class as having a finite carrier set `UNIV`. The predicate finite is predefined in the theory database, as well as the polymorphic set `UNIV` containing all elements for any type.

```
axclass
  fintype < type
  finite_set: finite (UNIV)
```

Next we define `fmaps` as an abbreviation for "partial" HOL functions whose domain is a finite type. The type variables α and β represent arbitrary types, but the `::` coerces α to be in the typeclass `fintype`.

```
  types (α, β)fmap = (α :: fintype) ~⇒ β (infixl "⇀" 50)
```

The construction $\alpha \overset{\sim}{\Rightarrow} \beta$ is again an abbreviation for $\alpha \Rightarrow (\beta$ `option`) where $\Rightarrow$ is the general type constructor for HOL functions and the option type is just the classical lifting known from domain theory.

```
  datatype α option = None | Some α
```

Since all HOL functions are total, the above map type $\alpha \overset{\sim}{\Rightarrow} \beta$ is the standard way to mimick partial functions: the constant `None` stands for undefined values.

Based on the definition of finite types we derive necessary infrastructure to support the use of finite maps in proofs. We need first to establish an induction scheme for finite types. Then using a representation of finite maps as finite sets of pairs that behave like functions, we derive the following induction scheme for the new generic type `fmap` from the induction scheme for finite sets by using a domain isomorphism between `fmap` and the set of pairs with function properties.

```
  ⟦ P empty;
    ⋀ x (F::label ↦ dB) y . ⟦  P F; x ∉ dom F ⟧ ⟹ P (F(x ↦ y))
  ⟧   ⟹ P F
```

The type `label` is a concrete finite type defined to represent a type for field names of objects in the following definition of ς-terms.

2.4 Binding with de Bruijn Indices

It is a known difficult problem how to represent binders when formalizing programming languages for meta-theoretical reasoning [17]. There are three alternatives that could be used in Isabelle/HOL: Higher Order Abstract Syntax (HOAS), de Bruijn indices, and Nominal Techniques. HOAS basically means identifying the binders of the object-logic with the binder of the meta-language, here the λ-binder of Isabelle/HOL. The advantage is that the full power of the theorem prover becomes immediately available for the object-language. However, this approach is out of the question for us, as it would rule out explicit reasoning about reduction and evaluation. The actual difficulty with a more direct representation of a binding construct is not to define it [2] but to work one's way around the problem of α-conversion. One very recent way of dealing with this problem is provided by Nominal Techniques [19]. Here, basically an implicit factorization over concrete variable names, using a so called "support" representing all possible permutations of variables, enables to abstract from concrete names of variables. We have experimented with a recent implementation of a package for Nominal Techniques in Isabelle/HOL [18], but had to find out that neither recursive datatypes nor fancier constructs like our `fmap` are currently being supported. [3]

Consequently, we had to use the classical technique of de Bruijn indices. De Bruijn indices overcome the problem of concrete variable names, and thus α-conversion, by simply eliminating them. A variable is replaced by a natural number that represents the distance — in terms of nesting depth — of this variable to its binder. Thereby terms contain only numbers, no variable; α-conversion becomes obsolete. This is a considerable advantage as α-conversion is a difficult problem both from a practical point of view and for mechanical proofs. An example for illustrating the use of de Bruijn indices is given by the following simple λ-term.

$$\lambda x.\lambda y.(\lambda z.\, x\, z)y = Abs(Abs(Abs(Var\, 2)\$(Var\, 0))(Var\, 0))$$

Note that, different variables may be represented by the same number, e.g., z and x both are $Var\, 0$. De Bruijn indices relieves one from having to deal with α-conversion: for example both $\lambda x.x$ and its α-equivalent $\lambda y.y$ are represented by $Abs(Var 0)$. The disadvantage of de Bruijn indices is that substitution, normally used for the definition of application, is difficult to construct. A term has to be "lifted", that is, his "variables" have to be increased by one, when it moves into the scope of an abstraction in the process of substitution.

3 The Theory of Objects in Isabelle/HOL

3.1 Formalizing ς-Terms

The type `dB` of ς-terms in Isabelle/HOL is given by the following datatype declaration. Note, that there are two types of labels: `label` represents the method descriptors in an object while `Label` is the type of aspect labels. Both actual types are just type synonyms for `nat`, the type of natural numbers in Isabelle/HOL.

```
datatype dB =  Var nat
            |  Obj (label ⇀ dB) type
            |  Call dB label
            |  Upd dB label dB
            |  Asp Label dB    ("⟨ _ ⟩")
```

[2] HOL is expressive enough to accommodate functions from variables to values as higher order parameter for a binding constructor.

[3] In fact, not even the datatype of lists is yet compatible with nominal techniques.

The constructor `Var` builds-up a new term `dB` from a `nat` representing the de Bruijn index of the variable. In the constructor `Obj` for objects we see now our `fmap` constructor being used: an object is recursively defined by a finite map from `label`, the predefined types of "field names", to arbitrary terms of type `dB`. The second argument of type `type` to the `dB`-constructor `Obj` is the Object's type. It will be formally introduced in Section 4. We insert the type with an object in order to render the typing relation unique (see Section 4). The cases `Call` and `Update` similarly represent, field selection and update of an object's field. The field constructor `Asp` enables the insertion of aspect labels into object terms. We do not assign any semantics to labels until we define weaving in Section 5. The annotation behind the constructor in quotation marks defines the mixfix syntax: we can use the notation `l⟨t⟩` as abbreviation for `Asp l t`.

Next, we need to define lifting and substitution in order to arrive at a reduction relation for our object-terms. These definitions are very technical, so we skip them here. For a full account see the Isabelle/HOL sources at the authors' web page [8]. The Isabelle/HOL mixfix syntax enables a definition of substitution for ς-terms as `t[s/n]` meaning *replace* `n` *by* `s` *in* `t`. We define a small step operational semantics by a relation $\rightarrow_\beta$ using an inductive definition.

```
inductive →β
intros
  beta: l ∈ dom f  f ⟹ Call (Obj f) l →β the(f l)[(Obj f)/0]
  upd : l ∈ dom f  ⟹ Upd (Obj f T) l a →β  Obj (f (l ↦ a) T)
  sel : s →β t ⟹ Call s l →β Call t l
  updL: s →β t ⟹ Upd s l u →β Upd t l u
  updR: s →β t ⟹ Upd u l s →β Upd u l t
  obj : ⟦ s →β t; l ∈ dom f ⟧
        ⟹ Obj (f (l ↦ s) T)  →β Obj (f (l ↦ t) T)
  asp : s →β t ⟹ l ⟨ s ⟩ →β l ⟨ t ⟩
```

The rules `sel`, `updL`, and `updR` merely encode that reduction can be performed in contexts. The others represent quite closely the original semantics of ς. (The substitution `[(Obj f T)/0]` in the rule `beta` replaces the self parameter for the outermost variable in the object's *lth* field `f l`. The operator `the` selects an α-element in an option datatype when it is defined, i.e. unequal to `None`. The cases `upd` and `obj` just replace inside objects. Additionally in some cases, the additional proviso `l ∈ dom f` assures that there is no call out of range of an object. The case `asp` enables, similar to the rules `sel`, `updL`, and `updL`, to evaluate in a labelled context. There is no other case for labels corresponding to the fact that no semantics is attached to labels until later.

This is the basic machinery for ς-terms in Isabelle/HOL with which we can represent any object term and evaluate it. The original notation used by Abadi and Cardelli (see Section 2.2) does not differ very much from our notation in Isabelle/HOL. The object $[l = \varsigma(x)x.l, n = \varsigma(x)x]$ is, for example, represented as `Obj(∅(l ↦Call(Var 0) l)(n ↦(Var 0))) T` for some suitable `T` where ∅ represents the empty map. It would be easy to add more syntactic sugar by defining additional mixfix syntax to achieve even closer resemblance.

The next important property to examine is determinacy of the evaluation.

3.2 Confluence

Confluence means that the reduction relation is deterministic. That is, whenever the reduction of an expression `x` of the language can return differing results `y` and `z` there are further reductions possible to a term `u` such that `x` and `y` can be further reduced to `u`. Formally, this property is based on the *diamond property* of a relation $\sim$.

$$\texttt{diamond}(\sim) \equiv_{df} \forall\ \texttt{x}\ \texttt{y}.\ \texttt{x} \sim \texttt{y} \longrightarrow \forall\ \texttt{z}.\ \texttt{x} \sim \texttt{z} \longrightarrow \exists\ \texttt{u}.\ \texttt{y} \sim \texttt{u} \wedge \texttt{z} \sim \texttt{u}$$

Confluence of a relation $\sim$ is defined as diamond($\sim^*$) where $*$ denotes the reflexive, transitive closure of a relation. We proved confluence of the reduction $\rightarrow_\beta$ in Isabelle/HOL.

Theorem 1 (Confluence of $\rightarrow_\beta$).

$\texttt{diamond}\ (\rightarrow_\beta^*)$

We were able to re-use the existing structure of the confluence proof from our earlier experiment [7], which used Nipkow's framework for the classical Church-Rosser proof as described by Barendregt in [3]. The classical trick already used in the application for the λ-calculus is to use a so-called *parallel reduction* $\rightarrow_{\|}$ for which the diamond property is true. Indeed, in general, the original reduction relation $\rightarrow_\beta$ does not verify `diamond` $\rightarrow_\beta$, and proving `diamond` $\rightarrow_\beta^*$ directly is very difficult. Thanks to the following theorem, we only have to show the inclusion of the parallel reduction relation in between the original reduction relation $\rightarrow_\beta$ and its transitive, reflexive closure.

$$[\![\ \texttt{diamond} \rightarrow_{\|};\ \rightarrow_\beta\ \subseteq\ \rightarrow_{\|};\ \rightarrow_{\|}\ \subseteq\ \rightarrow_\beta^*\ ]\!] \Longrightarrow \texttt{confluent} \rightarrow_\beta$$

Naturally, there were numerous adjustments to be made to the proofs in [7]. These were partly due to the – compared to plain lists – relative lack of prepared lemmas for finite maps. Especially the automatically generated induction schemas for datatypes using finite maps were not readily usable and had to be replaced by manually proved counterparts.

4 Type Safety for Objects

Generally, types in programming languages are a means to ensure statically as much soundness as possible. A type system defining types in an inductive style encodes therefore a decidable portion of the semantics of the language in question. Type safety entails that, whenever a program can be typed according to the type system, it fulfils the semantic property that is encoded in the type system. Classically, type systems encode the properties *progress* and *preservation* [20]. Progress describes the property that a well typed term is either a value or can be reduced further according to the evaluation relation. Preservation states that reduction does not change the type of a term, thereby ensuring that the evaluation does not endanger the semantic properties. When encoding a type system in Isabelle/HOL we implicitly prove the decidability of this type system by expressing the rules of the type system as rules of an inductive definition. This is a nice by-product of using a theorem prover.

The type system we define is derived from the original simple type system that Abadi and Cardelli presented in their work [2]. However, we could simplify it by omitting their "outcome" function which they use to describe definedness. Instead we use the explicitness readily available in our model: since finite maps are functions we can use the notion of "domain" to describe that a call is within range of an object. Since the ς-calculus does not contain values, we consider that a value is reached whenever a term is an object even though inside the object further reduction might be possible. This is as much as we can get because in the ς-calculus non-terminating objects may well be defined. For example, the ς-term $[l = \varsigma(x)x.l]$ enters a non-terminating reduction, and — what is more important for safety – reproduces itself. Hence, there is generally no progress unless we refrain from evaluation inside objects — as we will prove shortly.

In the ς-calculus every term is an object. Hence, the following recursive Isabelle/HOL datatype defines the possible types.

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

To access the actual type at a given label we define the following projection.

```
  (Object l)!n = the (l n)
```

The type system for ς-term is defined by an inductive typing relation. This relation `typing` is given as a set of triples containing the type environment, the term, and its type.

```
  typing  :: (type list × dB  × type) set
```

We use Isabelle's mixfix possibilities to define the syntax `env ⊢ x : T` conveniently annotating that term `x` has type `T` in environment `env`, or, more formally, `(env,x,T) ∈ typing`. Type environments, like `env`, are defined such that they can be simply extended using a stack operator that we defined for this purpose. For example, `env⟨0:A⟩` denotes the environment `env` extended with the type assumption that the outermost variable has type `A`.

Now, the inductive definition for the typing relation consists of the following rules.

```
inductive typing
intros
  T_Var : ⟦ x < length env; (env ! x) = T ⟧  ⟹ env ⊢ Var x : T
  T_Obj : ⟦ dom b = dom B; ∀ l ∈ dom B. env⟨0:B⟩ ⊢ the(b l) : B!l ⟧
                    ⟹ env ⊢ Obj b B :  B
  T_Call: ⟦ env ⊢ a : A; l ∈ dom A ⟧    ⟹ env ⊢ Call a l : A!l
  T_Upd : ⟦ env ⊢ a : A; l ∈ dom A ; env⟨0:A⟩ ⊢ n : A!l ⟧
                    ⟹ env ⊢ Upd a l n : A
```

The variables `A` and `B` range over types. The variable `env` represents a type environment containing type assumptions for variables. A type environment is a mapping from variables to types, its extension by a new assumption of "`x` has type `A`" is annotated as `env⟨x:A⟩` (where `x` is a natural number in our de Bruijn representation). The operator `!` is used for selecting the nth element of a type environment. It is already provided in Isabelle/HOL for selecting the nth element of a list. Note, that we also use it in an overloaded fashion as the projection for object types. The rule `T_Var` accesses the type environment `env` to ensure variable types. The rule `T_Obj` describes how an object's type is derived from its constituents. An object of type `B` is formed from bodies `the(b l)` of types `B!l` that may use the self parameter fixed as `0` in the type environment. When a method `l` is invoked on an object `a` of type `A` the result `Call a l` has type `A!l` (`T_Call`). Similarly an update of a method may take place in a position `l` of an object that has the right body type under the assumption of the self parameter (`T_Upd`).

Given this type system we prove type safety first for the ς-calculus in Isabelle/HOL. We prove the following two theorems.

Theorem 2 (Progress).

$$[\![\ \texttt{[]} \vdash \texttt{t : A};\ \nexists\ \texttt{c. t = Obj c}\ ]\!] \Longrightarrow \exists\ \texttt{t'. t} \rightarrow_\beta \texttt{t'}$$

The second theorem is the preservation theorem, sometimes also called *subject reduction.*

Theorem 3 (Subject Reduction).

$$[\![\ \texttt{env} \vdash \texttt{t : A; t} \rightarrow_\beta \texttt{t'}\ ]\!] \Longrightarrow \texttt{env} \vdash \texttt{t' : A}$$

The proofs of these theorems including the definition of the type system take about 800 lines of Isabelle code. We could gather some initial inspiration from the type safety proof for the typed λ-calculus that has been performed by Nipkow [14]. Although the preservation theorem for the λ-calculus has a lot in common with our case, the progress theorem differs already in its formulation (for the simply typed λ-calculus strong normalization is proved which includes termination). Consequently the proof for progress is quite different in the two formalizations.

We were also able to show the uniqueness of the types.

Theorem 4 (Uniqueness).

$$[\![\ \texttt{env} \vdash \texttt{t : T; env} \vdash \texttt{t : T'}\ ]\!] \Longrightarrow \texttt{T = T'}$$

This property would not hold true without the type annotation introduced in the initial datatype declaration. For example, the object $[l = \varsigma(x)x]$ would have types `Object (empty (l ↦ T))` *for any type* `T`, if we had not fixed the type inside the object. Accidentally, we would have introduced some kind of polymorphic functions. The proof of the uniqueness is rather straightforward and essentially consists of an induction over the typing relation.

5 Aspects, Weaving, and Types for Aspects

The ingredients of an aspect-oriented program are a base program written in an object-oriented language, and a set of aspects. The aspects consist of a selection of pointcuts and an advice that shall be applied at those points. The process of actually plugging in the advice at the specified pointcuts is called weaving. In this section we present these features in Isabelle/HOL for the ς-calculus together with a type system for aspects.

5.1 Aspects

An aspect can be simply defined as a selection of pointcuts and an advice. Since our model is in Higher Order Logic, where sets are isomorphic to predicates, we can assume that our selection of pointcuts is a set of labels. The advice is a ς-term not enclosed in an object, because an advice is applied to the sub-expression of a ς-program that is marked by a label returning another ς-term as a result. Hence, in Isabelle/HOL aspects can be simply defined as follows.

```
datatype aspect = Aspect (Label list) dB    ("⟨ _._ ⟩")
```

The first element is the pointcut set and the second element the advice to be applied to all points matching the pointcut description, i.e. being member of this set. The mixfix syntax at the righthandside enables the annotation of an aspect as ⟨L.a⟩.

5.2 Weaving

Given a base program in the ς-calculus readily labelled with aspect labels and given some aspects, the weaving function now only has to step through the term while applying the aspect. We consider this approach to resemble static weaving, but given the functional nature of our calculus, we consider the result to be valid for dynamic approaches as well. Therefore, we define a function "weave", represented as ⇓, that takes a ς-program and an aspect and returns a ς-program. The second operator weave_option is an auxiliary function that is needed to "map" the weaving function over the finite maps representing objects.

```
weave :: [ dB, aspect ] ⇒ dB    ("⇓")
weave_option :: [ dB option, aspect ] ⇒ dB option ("⇓opt")
```

We define the weaving function for the simple case of applying one aspect to a program. The general case is later derived by repeated application. The definition of the simple case is given below in a mutual recursive definition defining the semantics of weave and weave_option by simple equations. In case of weaving an aspect onto a variable Var n the advice has no effect. The case l⟨t⟩ is the interesting one because now the ς-term for aspects, Asp, is finally equipped with semantics. In case that the label is in the pointcut specified by the first component of the aspect, the aspect matches. Consequently, the advice part of the aspect `a` is applied to the current term `t`. Otherwise the aspect has no effect. The label is not eliminated during the weaving process to enable repeated weaving.

```
primrec
   (Var n) ⇓ ⟨L.a⟩ = Var n
   l ⟨ t ⟩ ⇓ ⟨L.a⟩ = if  l ∈ set(L) then l ⟨ a[(t ⇓ ⟨L.a⟩)/0] ⟩
                                    else l ⟨ t ⇓ ⟨L.a⟩ ⟩
```

The Isabelle/HOL projection `set` transforms a list (here, of labels) into the set of all elements contained in the list. Note, that the functional application of the advice `a` to the term `t` is realized using substitution for `0` using the same idea as in the rule `beta` of the reduction relation.

The next two equalities for `Call` and `Upd` simply define that the weave process is to be passed through to the corresponding sub-terms.

```
   (Call s l) ⇓ A = Call (s ⇓ A) l
   (Upd s l t) ⇓ A = Upd (s ⇓ A) l (t ⇓ A)
```

The primitive recursive equations defining the semantics for `Obj` is now the point where the recursion changes to the auxiliary operator `weave_option`. The auxiliary operator enables the pointwise definition of advice on the fields of the object by lifting the weaving function over the λ to argument position. In the defining equations for `weave_option` ($\Downarrow_{\mathrm{opt}}$) we see the benefit gained by using the option type: we can explicitly use pattern matching to distinguish the case for unused field labels (`None`) and actual object fields matching out the field value with `Some`.

```
   (Obj f T) ⇓ A = Obj (λ l. ((f l) ⇓opt A)) T
   None ⇓opt A = None
   (Some t) ⇓opt A = Some (t ⇓ A)
```

The generalization of the weaving function to lists of aspects is simply defined using some of the well-elaborated functors for lists available in Isabelle/HOL. The predefined functor `foldl` enables the application of a function repeatedly to an argument taking second arguments from a list. It is defined as follows.

```
consts  foldl :: (β ⇒ α ⇒ β) ⇒ β ⇒ α list ⇒ β
primrec
  foldl_Nil:  foldl f a [] = a
  foldl_Cons: foldl f a (x#xs) = foldl f (f a x) xs
```

This is exactly what we need: iterated application of weaving to a ς-term `t` using advice from a list of advice `l`.

```
   Weave t l ≡df delabel(foldl (op ⇓) t l)
```

The function `delabel` is a simple recursive function that deletes all labels from the weaving result thereby producing a "label-free" ς-calculus term. This final step is necessary to arrive at an unambigious term at the end of weaving. Otherwise we would have to consider equivalence classes of labelled terms.

5.3 Type System

We next present a type system for aspects. We have succeeded in designing this type system for aspects and proved type safety completely in the theorem prover Isabelle/HOL. Here, we introduce the major definitions and the proved theorems. The entire proof development in Isabelle/HOL is available on the authors' web-page [8].

The basic idea of the type system is that we attach types to aspect labels. Any advice that may be woven in at a particular point has to be conform to the type attached to this point's label. For type safety, we found an elegant way of proving that aspect weaving respects types. This general results grants to recover type safety for weaving from the previous type safety results for the typed ς-calculus (see Section 4). We extend this basic type system of our Isabelle/HOL formalization for objects of the ς-calculus. We use a second environment `L` – besides the basic type environment – to keep track of label types during the process of typing. Technically, we need to redefine the entire inductive definition for the type system, as an extension mechanism for inductive definitions is not provided in Isabelle/HOL.

Compared to the rules dealing with the existing, pure, ς-constructors in the `dB` datatype (cf. Section 4) the only notable change is that the environments now are complemented by a label environment `L`. Hence, the new inductive relation `typing` has four parameters. The additional label environment `L` maps labels to types. It enforces that a given label has the same type at all occurrences.

For instance, the `Var` case features now an additional environment `L` of label types.

```
T_Var : ⟦ x < length env; (env ! x) = T ⟧  ⟹ env,L ⊢ Var x : T
```

Similarly, the other three rules are identical to the rules for the pure ς-calculus (see Section 4), except for the additional parameter `L` as the label type environment in all typing judgements.

Finally, we add one new rule for the typing of labels. It states that a label has the type assigned in the environment and that a labelled term's type has to be conform to the label type. Given a term `a` of type `A` we can insert a label `l` in front of `a` if we are in the same environment.

```
T_lab : ⟦ l!i = A; i < length L; env,L ⊢ a : A ⟧ ⟹ env,L ⊢ l⟨ a ⟩ : A
```

The introduction of the second parameter has little impact on the proofs presented in the previous section. In particular, uniqueness is not weakened by the introduction and looks like this in its adapted form.

```
⟦ env, L ⊢ t : T; env, L' ⊢ t : T' ⟧ ⟹ T = T'
```

These are all additions we made to the type system. Still, one decisive information for a meaningful static analysis is missing. We know how to type aspect labels now and we know how to type labelled programs. However, in aspect-orientation, the process of weaving plays a rôle in the semantics. So, we need to lift the typing to the weaving operation. Since we added weaving not as a term constructor of the actual term language of labelled ς-terms `dB`, the typing for the weaving function is not part of the type system seen above.

But, as we are in HOL weaving is a function of the meta-level, i.e. Isabelle/HOL, and we can introduce the well-formedness of weaving also at the meta-level. Therefore, our expressivity is not lessened. First, we define a predicate that ensures that a set of pointcuts and an advice are compatible.

$$\texttt{wf\_adv L } \langle \texttt{ L. a}\rangle \equiv_{df} \forall \texttt{ l} \in \texttt{set(L). } \exists \texttt{ A. L!l = A} \wedge \texttt{ []}\langle \texttt{0: A}\rangle\texttt{, L} \vdash \texttt{a : A}$$

This predicate enforces that there must be one environment (empty base-types and some appropriate label-types) such that all labels in the pointcut set of the aspect can be typed according to the advice. Note, that an advice is thereby constrained to have identical input and output type. Further loosening of this constraint towards some kind of conform subtyping here is future work (see Section 6.3).

Given this internal well-formedness of aspects, we can lift it up to define well-formedness between an aspect and a base program.

$$\texttt{wf\_at L t a} \equiv_{df} \exists \texttt{ T. wf\_adv L a} \wedge \texttt{ [], L} \vdash \texttt{t: T}$$

This predicate can again be lifted to sets of aspects.

$$\texttt{wf L t A} \equiv_{df} \forall \texttt{ a} \in \texttt{set(A) . wf\_at L t a}$$

With all these preparations we are now able to identify a theorem that encodes the preservation of typing through weaving.

Theorem 5 (Weaving Preservation).

$$[\![\texttt{ wf L t A; [], L} \vdash \texttt{t : T }]\!] \Longrightarrow \texttt{[], L} \vdash \texttt{t} \Downarrow \texttt{A : T}$$

This theorem is the central theorem for type safety for aspects in our setting because the usual type safety theorems, progress and preservation, are simply implied by it.

Corollary 1 (Aspect Progress).

$$[\![\texttt{ wf L t A; [], L} \vdash \texttt{t A: T; } \nexists \texttt{ c B. t} \Downarrow \texttt{A = Obj c B }]\!] \\ \Longrightarrow \exists \texttt{ t'. t} \Downarrow \texttt{A} \rightarrow_\beta \texttt{t'}$$

Corollary 2 (Aspect Preservation).

$$[\![\texttt{ wf L t A; [], L} \vdash \texttt{t : T ; t} \Downarrow \texttt{A} \rightarrow_\beta \texttt{t' }]\!] \Longrightarrow \texttt{env, L} \vdash \texttt{t' : T}$$

6 Conclusions

In this paper we have presented a formalization of our theory of aspects in the theorem prover Isabelle/HOL. It consists of the ς-calculus with an extension by so-called labels for the representation of join-points and definitions of weaving functions. We have proved the confluence and type-safety of the basic language of ς-objects and for the extension to aspects.

6.1 Related work

There are some, partly still ongoing, strands of research concerning theoretical work for the support of aspects. The approach which is probably closest to ours is the work by Ligatti et al. [12]. We differ from their approach in that we use the ς-calculus as a basis, thus being object-oriented in the core-calculus, whereas they start from some λ-like functional language. Clifton and Leavens devised their MiniMao language [4] which is a typed aspect-oriented language based on a small imperative Java-subset. Another approach taken by Jagadeesan et al [9] concentrates on a more event based view on aspect-orientation. The authors use the π-calculus as a basis for the construction of an aspect-calculus.

However, none of the above mentioned theoretical accounts provides a mechanization in a theorem prover or similar tool. We are not aware that there are any attempts to formalize a theory of aspects inside a theorem prover. In particular, in the field of language semantics and type systems we consider definitions and proofs sufficiently complex to render automated proofs an imperative condition for high quality developments.

6.2 Compositionality and Run-Time Weaving

An important question for aspects and their practical usability is the compositionality of weaving. A similar question is whether run-time weaving is possible. Figure 3 illustrates this question graphically: when does this diagram commute? (index *sc* stands for source, *bc* for bytecode, *p* for program, and *ptc* and *adv* for pointcut and advice.) An immediate success of our formalization is that we

$$\begin{tikzcd}
(p_{sc}, ptc_{sc}, adv_{sc}) \arrow[r, "\texttt{weave\_sc}"] \arrow[d, "\langle \texttt{comp}, ptc_{comp}, \texttt{comp} \rangle"] & p'_{sc} \arrow[d, "\texttt{comp}"] \\
(p_{bc}, ptc_{bc}, adv_{bc}) \arrow[r, "\texttt{weave\_bc}"] & p'_{bc}
\end{tikzcd}$$

Fig. 3. do compile-time and run-time weaving commute?

have made one step towards identifying the conditions for a precise analysis for this question. In the aspect-calculus that we have presented in this paper we can state the corresponding compositionality proposition as follows.

$$[\![\texttt{ t} \rightarrow_\beta \texttt{t'; a} \rightarrow_\beta \texttt{a'}]\!] \Longrightarrow \texttt{t} \Downarrow \langle \texttt{ L. a } \rangle \rightarrow_\beta \texttt{t'} \Downarrow \langle \texttt{ L.a' } \rangle$$

Compilation is here replaced by interpretation because our calculus is functional.

6.3 Subtypes

As we have seen in Section 5.3 the type system for aspects is constrained to have identical input and output types. We are considering the extension of out simple type system with sub-types. This would, in principle, give the means to relax the constraints on aspect types. However, neither a contravariant nor a covariant type refinement is possible for aspects in general: counterexamples may be constructed (see Jagadeesan et al. [9]).

6.4 Discussion

We did not embed weaving as a first-class function into our term language. This might at first sight seem odd, but we do not need to make weaving first-class. As we are in Higher Order Logic, we can reason about a meta-level function over ς-terms also in the object-logic. It has proved to be, on the contrary, an advantage to formalize weaving as a meta-logical HOL-function because we did succeed, in addition, to express its semantics using primitive recursion. The fact that weaving is a HOL-function implicitly grants us many useful properties – function properties combined with a primitive recursive definition save us a lot of explicit proof work because they are well supported in Isabelle/HOL.

One part of any aspect-oriented language is an object-oriented language for writing the base program and the advice. Therefore we chose the Theory of Objects by Abadi and Cardelli [1] as a basis for our mechanized theory of aspects. Although this base language is fairly small, it does – similar to the λ-calculus – enable the construction of all object-oriented features. On the other hand, this choice preserves generality of our approach: we stay independent of any particular implementation language, say Java, when we consider features and their related properties. Certainly, it must be shown that a small core language like the ς-calculus and our extension to aspects are equivalent to realistic programming language. Therefore, we intend to follow the approach taken in [12], where a type-preserving compilation is finally added from a real-world aspect-oriented language to the core-calculus.

Apart from providing a theoretical calculus for aspect-orientation, that is moreover mechanically verified, we believe our work to contribute to the safe use of this paradigm for the adaptable systems of the future. Our formalisation is a tool to experiment with different language constructs using the mechanical proof support to verify gradually type-safety and other more advanced properties like non-interference. In addition, a formalization as constructive as ours enables extraction of executable programs. Thereby prototypical tools for compilers and type-checkers are provided as a handy by-product for testing and implementation reference.

References

1. Martín Abadi and Luca Cardelli. *A Theory of Objects.* Springer, 1996.
2. Martín Abadi and Luca Cardelli. *A Theory of Primitive Objects. Theoretical Aspects of Computing Software, TACS'94.* LNCS **789**: 296–320, Springer, 1994.
3. Hendrik Pieter Barendregt. *The Lambda Calculus, its Syntax and Semantics.* North-Holland, 2nd edition, 1984.

4. C. Clifton and G. Leavens. Minimao: Investigating the semantics of proceed. In *Foundations of Aspect-Oriented Languages, FOAL'05*, 2005.
5. Pavel Avgustinov et al. Semantics of Static Pointcuts in Aspect, *Principles of Programming Lanugages, POPL'07*. ACM Press, 2007.
6. R. Filman and D. Friedman. *Aspect-Oriented Programming is Quantification and Obliviousness.* In Workshop on Advanced Separation of Concerns, OOPSLA 2000, Minneapolis, USA, October 2000.
7. L. Henrio and F. Kammüller. A Mechanized Model of the Theory of Objects. *9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, FMOODS 2007*. LNCS **4468**, Springer, 2007.
8. S. Jähnichen and F. Kammüller. *Ascot: Formal, mechanical foundation of aspect-oriented and collaboration-based languages.* Web-page at `http://swt.cs.tu-berlin.de/∼flokam/ascot/index.html`. Project with the German Research Foundation (DFG), 2006.
9. R. Jagadeesan, A. Jeffrey, and J. Riely. Typed Parametric Polymorphism for Aspects. [15]: 267–296.
10. F. Kammüller. *Interactive Theorem Proving in Software Engineering.* Habilitationsschrift (habilitation thesis), Technische Universität Berlin, 2006.
11. F. Kammüller and M. Vösgen. Towards Type Safety of Aspect-Oriented Languages. In *Foundations of Aspect-Oriented Languages, FOAL'06*, 2006.
12. Jay Ligatti, David Walker and Steve Zdancewic. A type-theoretic interpretation of pointcuts and advice. [15]: 240-266.
13. L. C. Paulson. *Isabelle: A Generic Theorem Prover.* LNCS **828**, Springer, 1994.
14. Tobias Nipkow. More Church Rosser Proofs. *Journal of Automated Reasoning.* **26**:51–66, Kluwer Academic Publishers,2001.
15. *Science of Computer Programming: Special Issue on Foundations of Aspect-Oriented Programming.* **63**(3), Elsevier, 2006.
16. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, LNCS **2283**, Springer, 2002.
17. The POPLmark challenge. `http://alliance.seas.upenn.edu/ ∼plclub/ cgi-bin/ poplmark`. July 2007.
18. Christian Urban et al. Nominal Methods Group. Project funded by the German Research Foundation (DFG) within the Emmy-Noether Programme, 2006. Web-page at `http://www4.in.tum.de/∼urbanc/Nominal/`.
19. Christian Urban and Christine Tasson. Nominal Techniques in Isabelle/HOL. In *20th International Conference on Automated Deduction, CADE'05*. LNCS, **3632**, Springer, 2005.
20. A. Wright and M. Felleisen *A Syntactic Approach to Type Soundness.* Information and Computation, **115**: 38–94, Elsevier, 1994.

A Appendix: Definitions and Theorems

chapterntertheorem0 In this Appendix we present the core definitions and theorems of our aspect calculus. For complete proofs, please see our project's homepage `http://swt.cs.tu-berlin.de/~flokam/ascot`

Finite Maps

The mechanisation of the ς calculus is based on a novel concept of finite Maps. We show the theorems and definitions used in the formalization of that construct.

Definition 1 (Finite Maps).

```
axclass fintype < type
finite_set: " finite (UNIV)"
types (α, β)fmap = "(α :: fintype) ⇀ β" (infixl "⇀" 50)
```

Theorem 6 (Induction over finite sets).

```
P  ⟹ (⋀ x (F :: (α :: fintype)set). x ∉ F
       ⟹ P F ⟹ P (insert x F)) ⟹ P F;
```

Theorem 7 (Uniqueness of finite maps).

```
x = y ⟹ (f :: (α, β)fmap) x = f y
```

Theorem 8 (Cases distinction for finite maps).

```
(F :: (α ⇀ β)) = empty ∨ (∃ x y (F' :: (α ⇀ β)). F = F'(x ↦ y))
```

Theorem 9 (Finiteness of finite sets).

```
 finite (F :: ((α :: fintype) set))
```

Theorem 10 (Induction over finite maps).

```
P (λ x. None)
     ⟶ (∀ (F :: ((α :: fintype) ⇀ β)) x z . x ∉ dom F
     ⟶ P F ⟶ P (F (x ↦ z)))
     ⟶ P (F' :: (α ⇀ β))
```

Core Calculus

In this section we will introduce the basic extended ς-Calculus.

Lemma 1 (Labels are in fintype).

```
finite (UNIV :: (Label set))
```

Definition 2 (ς–terms).

```
datatype type =  Object "Label ⇀ type"

datatype dB =
    Var nat
  | Obj "Label ⇀ dB" type
  | Call dB Label
  | Upd dB Label dB
  | Label nat dB  ("_⟨ _⟩")
```

Definition 3 (Lifting).

```
consts
  lift        ::  "[dB, nat] ⇒ dB"
  lift_option ::  "[nat, dB option] ⇒ dB option"

primrec
  liftVar:    "lift (Var i) k = (if i < k then Var i else Var (i + 1))"
  liftCall:   "lift (Call a l) k = Call (lift a k) l"
  liftUpd:    "lift (Upd a l b) k = Upd (lift a k) l (lift b (k + 1))"
  liftObj:    "lift (Obj f T) k = Obj (λ l. (lift_option (Suc k) (f l))) T"
  liftLabel:  "lift (asp_la⟨ t⟩) k  = asp_la⟨(lift t k)⟩ "
  lift_None:  "lift_option k None = None"
  lift_Some:  "lift_option k (Some t) = Some (lift t k)"
```

Definition 4 (Substitution).

```
consts
  subst ::        "[dB, dB, nat] ⇒ dB"  ("_[_'/_]" [300, 0, 0] 300)
  subst_option :: "[nat, dB, dB option] ⇒ dB option"

primrec
  subst_Var:   "(Var i)[s/k] =
                (if k < i then Var (i - 1) else if i = k then s else Var i)"
  subst_Call:  "(Call a l)[s/k] = (Call (a [s/k]) l)"
  subst_Upd:   "Upd a l b [s/k] = (Upd (a [s/k]) l (b [lift s 0 / k+1]))"
  subst_Obj:   "Obj f T [s/k] = Obj (λ l. (subst_option (Suc k)(lift s 0)(f l))) T"
  subst_Label:  "(asp_la⟨ t⟩) [s/k] = asp_la⟨(t[s/k])⟩"
  subst_None:  "subst_option n s None = None"
  subst_Some:  "subst_option n s (Some t) = Some (t [s/n])"
```

Definition 5 (Reduction Relation).

```
consts
  beta :: "(dB × dB) set"

syntax
  "_beta" :: "[dB, dB] ⇒ bool"  (infixl "→β" 50)
  "_beta_rtrancl" :: "[dB, dB] ⇒ bool"  (infixl "→β*" 50)

translations
  "s →β t" ≡df "(s, t) ∈ beta"
  "s →*β t" ≡df "(s, t) ∈ beta*"

inductive beta
  intros
    beta [simp, intro!]: "l : dom f ⟹
                          Call (Obj f T) l →β  (the(f l))[(Obj f T)/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 →β  Call t l"
    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⟩"
```

Theorem 11 (Induction on ς–terms).

```
"⟦ ⋀ n. P1 (Var n);
⋀ f T. P3 f ⟹ P1 (Obj f T);
⋀ d l. P1 d ⟹ P1 (Call d l);
⋀ 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 ))"
```

Confluence of the Reduction

The actual proof of the confluence property for the reduction relation was performed using Nipkow's framework. As such it is not introducing new theorems, but proves the diamond property in pre-defined steps. The actual proofs can be found on the project homepage.

Definition 6 (Parallel Reduction Relation).

```
consts
  par_beta :: "(dB × dB) set"

syntax
  par_beta :: "[dB, dB] ⇒ bool"  (infixl "⇒" 50)
translations
  "s ⇒ t" ≡df "(s, t) ∈ par_beta"

inductive par_beta
  intros
    var  : "Var n ⇒ Var n"
    obj  : "⟦ dom s = dom s'; ∀  l ∈ dom s . the (s l) ⇒ the (s' l) ⟧
            ⟹ Obj s B ⇒ Obj s' B"
    upd  : "⟦ s ⇒ s'; t ⇒ t' ⟧ ⟹ Upd s l t ⇒ Upd s' l t'"
    upd' :"⟦ Obj s B ⇒ Obj s' B; t ⇒ t'; l ∈ dom s ⟧
                      ⟹ (Upd (Obj s B) l t)  ⇒ (Obj (s'(l ↦ t')) B)"
    sel  : "s ⇒ t ⟹ Call s l ⇒ Call t l"
    beta : "⟦ Obj f B ⇒ Obj f' B; l ∈ dom f' ⟧
            ⟹ Call (Obj f B) l ⇒ (the (f' l))[(Obj f' B)/0]"
    lab' : "⟦ t ⇒ t' ⟧  ⟹ i⟨ t⟩  ⇒  i⟨ t'⟩ "
```

Typing

This section presents the typing system for our core calculus and the basic type–safety theorems.

Definition 7 (Environment Operator).

```
constdefs
   shift :: "(α list) ⇒ nat ⇒ α ⇒ α list"    ("_⟨_:_⟩" [90, 50, 0] 91)
   "l⟨ i:a⟩ ≡df list_insert l i a"
```

Definition 8 (Notations).

```
constdefs
consts
  type_get :: "type ⇒ Label ⇒ type"  ("_^_" 100)
primrec
 " (Object l)^n = the (l n) "

consts
  type_get_opt :: "type ⇒ Label ⇒ type option"  ("_|_")
primrec
 " (Object l)|n =  (l n) "

consts
  do :: "type ⇒ (Label set)"
primrec
  "do (Object l) = (dom l)"
```

Definition 9 (Typing Relation).

```
consts
  typing :: "((type list) × type list  × dB  × type) set"
  typings :: "(type list) ⇒ type list ⇒  dB  list ⇒ type list ⇒ bool"

syntax (xsymbols)
  "_typing" :: "(type list) ⇒ (type list) ⇒  dB ⇒ type ⇒ bool" ("_,_ ⊢ _ : _")

translations
  "env,lab ⊢ t : T" ⇌ "(env, lab, t, T) ∈ typing"
  "env,lab ||- ts : Ts" ⇌ "typings env lab ts Ts"
```

```
inductive typing
 intros

    T_Var  : "⟦ x < length env; (env!x) = T ⟧ ⟹ env,L ⊢ Var x : T"
    T_Obj  : "⟦ dom b = do B; ∀ l ∈ (do B). env⟨ 0:B⟩,L ⊢ (the (b l)):((B^l)) ⟧
               ⟹ env,L ⊢ (Obj b B) :  B"
    T_Call : "⟦ env,L ⊢ a : A; l ∈ do A ⟧ ⟹ env,L ⊢ (Call a l)  : (A^l)"
    T_Upd  : "⟦ env,L ⊢ a : A; l ∈ do A ; env⟨ 0:A⟩,L  ⊢ n : (A^l) ⟧
               ⟹ env,L ⊢ (Upd a l n) : A"
    T_lab  : "⟦ env,L ⊢ a : A; i < length L; L!i = A⟧ ⟹ env,L ⊢ i⟨ a⟩  : A"
```

Lemma 2 (Substitution Lemma).

```
  "⋀ Ts. i ≤ length e ∧  e,L ⊢ u : T ⟹ e⟨ i:T⟩ ,L ||- ts : Ts ⟹
     e,L ||- (map (λ t. t[u/i]) ts) : Ts"
```

Theorem 12 (Subject Reduction).

"e,L ⊢ t : T ⟹ (⋀ t'. t $\to_\beta$ t' ⟹ e,L ⊢ t' : T)"

Theorem 13 (Subject Reduction *).

"t $\to_\beta^*$ t' ⟹ e,L ⊢ t : T ⟹ e,L ⊢ t' : T

Definition 10 (Label Functions).

```
consts
  delabel :: "dB ⇒ dB"
  delabel_option :: "[ dB option ] ⇒ dB option"
primrec
   "delabel  (Var n) = Var n"

   "delabel (l ⟨ t ⟩ )  = delabel t"
   "delabel (Call s l)  = Call (delabel s) l"
   "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)"

consts
  nolabel :: "dB ⇒ bool "
  nolabel_option :: "[ dB option ] ⇒ bool"
primrec
   "nolabel  (Var n) = True"
   "nolabel (l ⟨ t ⟩ )  = False"
   "nolabel (Call s l)  = nolabel s"
   "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 3 (Type Preservation of label removal).

```
"e,L ⊢ t : A ⟹ e,L ⊢ delabel t : A"
```

Theorem 14 (Uniqueness of Types).

```
"⟦ env, L ⊢ a : T; env, L' ⊢ a : T'⟧ ⟹  T = T'"
```

Theorem 15 (Progress).

```
"⟦ (∃ t' . t = delabel t'); [],L ⊢ t : A; ¬ (∃ c . ( t = Obj c A))⟧
        ⟹ (∃ b . ( t →β b)"
```

Aspects

This section presents the definition of aspects in our formalization. It introduces the datatype as well as the semantics of weaving and our notion of well–formedness. Finally it includes a few select theorems showing that the aspects do not harm typing.

Definition 11 (Aspect Datatype).

```
datatype aspect = ASP "nat list"  dB ("_._")

consts
  pc :: "aspect ⇒ nat list"
  adv :: "aspect ⇒ dB"
primrec
  "pc (ASP poc advi) = poc"
primrec
  "adv (ASP poc advi) = advi"
```

Definition 12 (Weaving).

```
consts weave :: "[ dB, aspect ] ⇒ dB"
   weave_option :: "[ dB option, aspect ] ⇒ dB option"
primrec
   "weave (Var n) a = Var n"

   "weave (l ⟨ t ⟩ ) a = (if  (l mem  (pc a))
                then (l ⟨ (adv a)[(weave t a )/0]⟩ )
                else (l ⟨ (weave t a) ⟩ ))"
   "weave (Call s l) a = Call (weave s a) l"
   "weave (Upd s l t) a = Upd (weave s a) l (weave t a)"
   "weave (Obj f B) a = Obj (λ  l. (weave_option (f l) a)) B"
   "weave_option None a = None"
   "weave_option (Some t) a = Some (weave t a)"
```

Definition 13 (Well–formedness of an Aspect).

```
constdefs wf_adv :: "[ type list, aspect] ⇒ bool"
          "wf_adv  L a ≡df  (∀ l. (l mem (pc a))
                         ⟶ (∃ A. ((((L!l) = A)
                                 ∧ (([]<0:A>),L ⊢ (adv a): A)))))"
```

Definition 14 (Aspect–base compatibility).

```
constdefs wf_at :: "[type list, dB, aspect] ⇒ bool"
    "wf_at L t a ≡df ∃  T. wf_adv  L a ∧  [], L ⊢ t: T"
```

Definition 15 (Lifting for all Aspects).

```
constdefs wf :: "[type list, dB, aspect list] ⇒ bool"
    "wf L t A ≡df ∀  a ∈  (set A). wf_at L t a"
```

Theorem 16 (A Well–formed Aspect preserves).

```
"⋀ env L A . ⟦ wf_adv  L A; [], L ⊢ t : T ⟧
          ⟹ [], L ⊢ weave t A : T "
```

Theorem 17 (Weaving does not harm Preservation).

```
‘‘⟦ wf L t A; [], L ⊢ t : T ; Weave t A →β t’ ⟧ ⟹ e, L ⊢  t’: T’’
```

Theorem 18 (Weaving does not harm Progress).

```
"⟦ wf L t A;  [], L ⊢  t: T; ¬ (∃ c . delabel (Weave t A) = Obj c T) ⟧
                         ⟹ (∃ t’. delabel (Weave t A) →β t’)"
```