Karlsruhe Institute of Technology

Semantische Objektorientierte Programmierung

Bachelorarbeit von

Florian Johannes Weber

an der Fakultät für Informatik
Institut für Intelligente Prozessautomation und Robotik (IPR)

Erstgutachter:	Prof. Dr. Heinz Wörn
Zweitgutachter:	Prof. Dr. Björn Hein
Betreuender Mitarbeiter:	Dr. Andreas Bihlmaier

November 2015 – März 2016

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Änderungen entnommen wurde.

Karlsruhe, den 29. März 2016

. . . . . .

(Florian Weber)

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation

Unter Ontologien versteht man in der Informatik eine Spezifikation einer Konzeptualisierung [3]. Die Integration in existierende Programmiersprachen ist bisher leider vergleichsweise limitiert: Statt Daten der Ontologie mit Objekten der Sprache eng zu verknüpfen, dienen existierende Lösungen oft eher der Steuerung und Verwaltung einer Ontologie als sich in diese eng zu integrieren.

Das Ziel der Arbeit ist zu untersuchen inwieweit eine engere Integration möglich und sinnvoll ist. Hierzu werden Objekte einer modernen Programmiersprache (C++) so in die Ontologie eingebunden, dass sie dort direkt als Individuen vorliegen über die im Anschluss Anfragen gestellt werden können.

1.2 Hintergrund

Bei domänenspezifischen Sprachen handelt es sich um Computersprachen welche für einen spezifischen Anwendungsbereich entworfen wurden und auf diesem oft äußerst ausdrucksstark und angenehm zu verwenden sind, dafür aber oft große Abstriche bei allgemeineren Aufgaben machen. Der in dieser Arbeit untersuchte Ansatz konzentriert sich im Gegensatz hierzu darauf, zwar einen vergleichbaren Level an Benutzerfreundlichkeit zu bieten aber sich insbesondere gut in eine allgemeine Programmiersprache zu integrieren ohne den Nutzer hierbei spürbaren Grenzen zwischen regulärer Programmierung und äußerst mächtiger Anfragen an einen Theorembeweiser auszusetzen. Während existierende Implementierungen ihre Klassen an einer bestehenden Ontologie orientieren und versuchen diese in die jeweilige Programmiersprache abzubilden, wird hier der umgekehrte Ansatz untersucht, vorhandene Objekte einer Sprache in der Ontologie abzubilden.

Traditionell lassen sich die meisten Programmiersprachen in imperative und deklarative Sprachen einteilen, wobei erstere beschreiben was ein Programm *tun* soll, während letztere beschreiben, was das *Ergebnis* einer Operation sein soll.

Im Rahmen dieser Arbeit wurde untersucht, wie sich diese Ansätze so miteinander verbinden lassen, dass ihre Vorteile erhalten bleiben, die Verbindung aber nicht wie ein Fremdkörper wirkt.

Zielsetzung ist weniger die Integration ontologischer Beziehungen in das Typsystem der Sprache, sondern vielmehr die Abbildung komplexer Beziehungen die erst zur Programmlaufzeit bekannt sind und daher in statisch typisierten Sprachen kaum mehr sinnvoll in das Typsystem integriert werden können (in dynamischen Sprachen wäre dies zwar eher möglich, aber vor dem Hintergrund des per Definition nicht vorhandenen statischen Typsystems bedeutungslos).

Ein existierender Versuch eine gewisse Form der direkten Abbildung von Logik in eine allgemeine Programmiersprache durchzuführen ist die Sprache Prolog. Diese ermöglicht es Probleme sehr leicht zu beschreiben und im Anschluss anfragen zu stellen:

```
% Source: https://en.wikipedia.org/wiki/Prolog
mother_child(trude, sally).

father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).

sibling(X, Y)      :- parent_child(Z, X), parent_child(Z, Y).

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).
```

Wenn man nach dieser Definition nun den Prolog-Interpreter fragt, ob die Aussage „`sibling(sally, erica)`“ wahr ist, wird dieser die Frage bejahen, da diese Aussage zwingend aus den obigen Axiomen folgt.

Die Integration in eine universelle Programmiersprache („General Purpose Programming Language“) verspricht nun die Probleme von Sprachen wie Prolog zu vermeiden, welche zwar bestimmte logische Probleme wie das obige sehr elegant lösen können, aber bei Problemen anderer Formen sehr schnell sehr unintuitiv werden können. So ist es etwa in vielen Fällen nur schwer möglich Algorithmen konkret zu definieren, da die letztlich stattfindenden Operationen durch den Abstraktionslevel leicht verschleiert werden können. Durch die Integration in eine allgemeine Programmiersprache wird diese Problematik vermieden indem jedes Problem auf eine angemessene Weise formuliert werden kann, ohne durch die Limitierungen der einen oder der anderen Herangehensweise zu einem schwer verständlichen Programm zu führen.

1.3 Zielsetzung

Ein Anwendungsbeispiel ist ein Programm für die Erstellung von Konferenzplänen: Für gegebene Mengen von Referenten, Vortragsräumen, möglichen Themen und speziellen Anforderungen (bestimmte Vorträge könnten zum Beispiel bestimmte Räume benötigen) kann es sehr schwer sein, eine Lösung zu finden und aufwendig einen geeigneten Algorithmus zu entwerfen und zu implementieren. Existierende Beweiser sind zwar gerade dafür geschrieben, solche Probleme in Angriff zu nehmen, die Konvertierung des Problems und die Anbindung an eine existierende Code-Basis können jedoch leicht ebenso aufwendig sein.

Das Ziel dieser Arbeit ist es, die Vorteile beider Ansätze zu vereinen ohne dafür die Nachteile in Kauf nehmen zu müssen.

Beispielhafter Code könnte dann so aussehen:

```
// create an ontology:
ontology o{ /* ... */ };

// create a speaker from some raw data:
```

```
speaker s{o, raw_data};

// create a talk from a speaker and a title:
// (the talk-ctor should be implemented in a way
// that will tell directly to the ontology who
// the speaker is)
talk t{o, s, "Semantic Object Oriented Programming"};

// room_and_slot will be a tuple of const references
// to a room and a slot that can be used by the
// talk without collisions
const auto room_and_slot = o.request_entities<room, slot>(
    assignment(s, t, r, sl), r, sl);
std::cout << "The talk " << s << " will happen in room"
          << std::get<0>(room_and_slot) << " in slot "
          << std::get<1>(room_and_slot) << ".\n";
```

Naturgemäß wäre für den praktischen Einsatz eine Axiomatisierung der Problemstellung nötig, die hier nicht gezeigt wird. Abgesehen davon handelt es sich in diesem Beispiel jedoch um eine nur marginal geänderte Anfrage eines funktionierenden realen Demonstrationsprogramms.

Zwar kann ein derartiger Ausschnitt noch nicht die Mächtigkeit der hier entwickelten Lösung vermitteln, allerdings sollte er bereits ein Gefühl für die Vorteile dieses Ansatzes geben und aufzeigen dass die Benutzung weder in C++ als Fremdkörper erscheinen muss, noch eine allzu außergewöhnliche Syntax für Prädikatenlogik umfasst.

1.4 Gliederung der Arbeit

Die verbleibenden Kapitel sind wie folgt gegliedert:

- Kapitel 2, „Grundlagen“, gibt einen groben Überblick über Templatemetaprogrammierung in C++ und eine Einführung in die Frage was Ontologien sind und wofür sie benutzt werden können.
- Kapitel 3, „Untersuchte Ansätze“, beschreibt und bewertet einige Techniken die im Rahmen dieser Arbeit entwickelt wurden, aber letztlich aus verschiedenen Gründen nicht benutzt wurden, sowie ein sehr kurze Einführung in einzelne verwendete Techniken, die bereits hier für das Verständnis nötig sind.
- Kapitel 4, „SOOP“ enthält eine ausführliche Beschreibung des letztlich benutzten Ansatzes zur **S**emantischen **O**bjekt**O**rientierten **P**rogrammierung.
- Kapitel 5, „Evaluation“ beschreibt eine beispielhafte Verwendung der entwickelten Bibliothek um ein reales Anwendungsproblem zu lösen.
- Kapitel 6, „Diskussion und Ausblick“ beurteilt das Endergebnis der Arbeit und gibt eine Liste von Vorschlägen in welche Richtung sich weitere Forschung anschließen könnte.

2 Grundlagen

2.1 C++

C++ ist eine statisch typisierte, üblicherweise zu nativem Maschinencode übersetzte, Programmiersprache, zu deren großen Vorzügen die Möglichkeit gehört Abstraktionen zu erstellen, die keinen Performance-Nachteile zur Programmlaufzeit haben („Zero Overhead Abstraction“). Nicht zuletzt deswegen erfreut sie sich bis heute sowohl in der Industrie, als auch im akademischen Umfeld großer Beliebtheit. Um dies zu erreichen ist die sogenannte „Templatemetaprogrammierung“ (TMP) ein wichtiges Stützglied. Es handelt sich hierbei um eine, ursprünglich nicht vorgesehene, Benutzung von Templates (dem Hauptwerkzeug für generische Programmierung in C++) um während des Übersetzungsvorgangs Werte zu berechnen und Ausführungspfade auszuwählen.

Die Grundidee ist hierbei, dass rekursive Funktionen als Klassentemplates geschrieben werden können, die eine Konstante definieren, welche von den Templateargumenten und einer anderen Instanziierung des selben Templates abhängen um schließlich über explizite Spezialisierungen eine Abbruchbedingung zu schaffen. Das Ergebnis ist, dass der Wert dieser Konstante nun zur Übersetzungszeit verfügbar ist:

```
template<unsigned Base, unsigned Exp>
struct exp {
    // recursive definition:
    enum { value = Base * exp<Base, Exp-1>::value };
};

// base-case:
template<unsigned Base>
struct exp<Base, 0> {
    enum { value = 1 };
};
```

In diesem Beispiel wird eine Metafunktion definiert, die Potenzen berechnet. Hierzu erhält sie zwei Ganzzahlen (`unsigned`) und definiert eine Konstante (hier als `enum`) namens `value` welche als das Produkt des ersten Arguments und dem Wert der rekursiven Instanziierung definiert wird. Im Anschluss wird der Basisfall (Exponent = 0) über eine teilweise Spezialisierung definiert.

Während dieses Konzept bereits sehr mächtig ist, wurde seit dem Erkennen der Bedeutung dieser Art der Programmierung sehr viel weitere Funktionalität geschaffen um die Arbeit mit der Technik zu erleichtern. So lässt sich obiges Beispiel etwa auch so implementieren:

```
template<unsigned N>
using num = std::integral_constant<unsigned N>;
```

```
template<unsigned Base, unsigned Exp>
struct exp: num<Base * exp<Base, Exp-1>::value> {};

template<unsigned Base>
struct exp<Base, 0>: num<1> {};
```

Im Gegensatz zu obigem Beispiel wird hier die Konstante nicht direkt definiert, sondern über das Standardbibliothekstemplate `std::integral_constant` und Vererbung zur Verfügung gestellt. Hierdurch wird das Beispiel kürzer und intuitiver zu verstehen. Zwar existiert mit `constexpr` mittlerweile auch eine oft überlegene Alternative zur Berechnung von Konstanten, jedoch ist es nur mit Templatemetaprogrammierung möglich, Typen zu berechnen oder Ausführungspfade zu parametrisieren.

Um nun komplexere Abbruchbedingungen oder logische Verzweigungen zu erschaffen sind für die Spezialisierung nicht nur einfache Konstanten möglich, sondern auch komplexere Ausdrücke welche ihrerseits von Template-Parametern abhängen. So ermöglicht das folgende Programm etwa die Überprüfung, ob es sich bei einem Typen um eine Instanz des Klassentemplates `std::pair` handelt:

```
template<typename T>
struct is_pair: std::false_type {};

template<typename L, typename R>
struct is_pair<std::pair<L, R>>: std::true_type{};

// remove reference and CV-qualifiers before check:
template<typename T>
constexpr bool is_pair_v = is_pair<std::decay_t<T>>::value;
```

Die Funktionsweise dieses Beispiels ist wie folgt: Zunächst wird ein Template `is_pair` definiert, welches durch das Erben von `std::false_type` (= `std::integral_constant<bool, false>`) signalisiert, dass sein Argument kein „pair“ ist. Im Anschluss wird das Template teilweise spezialisiert, so dass Instanzen von `std::pair` stattdessen über eine Version welche von `std::true_type` erbt implementiert werden. Hierbei handelt es sich um eine Form von „Pattern Matching“ wie man sie aus anderen Programmiersprachen kennt und nahezu beliebig komplexe Strukturen destrukturieren kann.

Templates können darüber hinaus als Parameter nicht nur integrale Datentypen und Typen entgegennehmen, sondern auch Templates (sogenannte „template-template-Parameter“), wodurch das entgegennehmende Template zu einem Template höherer Ordnung wird. Da Templates höherer Ordnung jedoch auch Templates sind, ist es naturgemäß auch möglich, ein Template zu schreiben, welches Templates höherer Ordnung als Parameter verlangt.

Ein einfaches Beispiel für ein Template welches ein Template als Argument entgegennimmt wäre etwa das folgende:

```
template<
    template<typename, typename> class Combine,
    typename Lhs,
```

```
    typename Rhs>
using combine_with = typename Combine<Lhs, Rhs>::type;
```

Während es mit Bezug auf Templates höherer Ordnung eher selten nötig ist, Parameter entgegenzunehmen die mehr als normale Templates sind, ist es vergleichsweise oft notwendig, eine variadische Anzahl von Templateparametern mit dem selben Metatyp entgegenzunehmen. Hierfür können variadische Templates benutzt werden, welche mehrere Argumente in sogenannten „Parameter-Packs“ speichern, die dann mit „...“ expandiert werden können.

All diese Techniken lassen sich kombinieren um zum Teil recht komplexe und mächtige Konstrukte zu errichten; im Folgenden etwa ein Beispielprogramm, welches über `check_is_pod` sicherstellt, dass alle Templateparameter die Anforderungen an sogenannte „Plain Old Datatypes“ erfüllen. Hierbei sei auch auf das Zusammenspiel mit Templates höherer Ordnung hingewiesen, die es ermöglichen, die eigentliche Logik nur ein einziges Mal generisch zu implementieren:

```
template<template<typename>class Check, typename T>
struct check_one {
    static_assert(Check<T>::value, "");
    using type = void; // in order to force instanciation
};

template<typename...>
class ignore_all{};

template<template<typename>class Check, typename...Ts>
struct check_all {
    using dummy = ignore_all<typename check_one<Check, Ts>::type...>;
};

template<typename...Ts>
using check_is_pod : check_all<std::is_pod, Ts...>;

int main() {
    (void) check_is_pod<int, double, float>{};
    // (void) check_is_pod<int, std::string, float>{}; // error
}
```

Diese Techniken sind für viele Anwendungen mehr als ausreichend, allerdings lösen sie nicht die Problematik, dass Templates nicht dazu in der Lage sind, Anforderungen an ihre Typargumente zu stellen. Zwar wird dies prinzipiell durch die „Technical Specification“ zu „Concepts“ gelöst, allerdings ist die praktische Verfügbarkeit geeigneter Compiler zu diesem Zeitpunkt nicht gegeben.

Eine Möglichkeit derartige Funktionalität dennoch zu bekommen, ergibt sich durch die Regel, dass Fehler in einer Templateinstanziierung nicht zwingend Kompilationsfehler sind (SFINAE, „Substitution failure is not an error“). Durch die gezielte Einführung überladener Funktionen mit unterschiedlicher Präferenz lässt sich hiermit die Gültigkeit beliebiger Ausdrücke überprüfen:

```
template<typename T>
struct require {
    using type = std::true_type;
};

template<typename T>
auto require_has_push_back_int_impl(T arg)
-> typename require<decltype(arg.push_back(0))>::type;

// Due to the varargs this function has
// the lowest possible precedence, which
// means that it is only taken if nothing else works:
std::false_type require_has_push_back_int_impl(...);

template<typename T>
using require_has_push_back_int
    = decltype(require_has_push_back_int_impl(std::declval<T>()));


int main() {
    // Prints whether the type has a method
    // push_back() that takes an int:
    std::cout
        << require_has_push_back_int<std::vector<int>>::value
        << require_has_push_back_int<int>::value
        << '\n'; // prints: 10
}
```

Zu beachten ist hier, dass keine der Funktionen jemals definiert wird. Dies stellt sicher, dass sie nie aufgerufen werden und jeder Versuch es doch zu tun in einem (gewünschten) Linker-Fehler endet.

Hierbei handelt es sich also um einen äußerst mächtigen Mechanismus der große Mengen von Problemen lösen kann. Obwohl mittlerweile auch viele Mechanismen in der Standardbibliothek existieren um diese Art der Programmierung angenehmer zu gestalten, ist es jedoch so, dass oftmals elegantere Methoden existieren (etwa normale Funktionsüberladung). Die Grenzen verlaufen hier jedoch fließend, da ja auch SFINAE letztlich auf Funktionsüberladung setzt.

Für ein weiteres Beispiel in dem SFINAE eine elegante Lösung darstellt, sei auf den folgenden Quelltext verwiesen, der demonstriert wie ein Funktionstemplate je nach Typargument verschieden implementiert werden kann, etwa wenn eines der Argumente ein „Plain Old Datatype“ (POD) ist:

```
template<
    typename T,
    std::enable_if_t<std::is_pod<T>::value, int> = 1
>
void some_fun(T arg) {
    // POD version
```

```
}

template<
    typename T,
    std::enable_if_t<!std::is_pod<T>::value, int> = 1
>
void some_fun(T arg) {
    // general version
}
```

Die Funktionsweise des obigen Codes ist wie folgt: `std::enable_if` nimmt zwei Argumente, wobei das erste ein boolscher Wert und das Zweite optional ist. Falls das erste Argument wahr ist und auch nur dann, erhält die Templateinstanz einen Typalias namens `type` für den Typ des zweiten Templatearguments oder, falls keines agegeben wurde, `void`. Das im Beispiel benutzte `std::enable_if_t` ist nichts anderes als ein Alias für `typename std::enable_if<...>::type`.

Wird nun `some_fun` mit einer Instanz eines Typen `T` aufgerufen, so werden zunächst beide Versionen der Funktion in Erwägung gezogen. Um zu sehen, welche geeignet ist, werden daher beider probeweise instanziiert. O.B.d.A. wird nun angenommen, T sei ein POD. In diesem Fall wird der Versuch das zweite Template zu instanziieren dazu führen, dass versucht wird, den Wert von `std::enable_if<false, int>::type` zu lesen welcher allerdings nicht existiert. Daher scheitert die Instanziierung des zweiten Templates und es verbleibt nur die erste Version, welche instanziiert werden kann, da `std::enable_if<true, int>::type` ja existiert, wodurch der zweite Templateparameter hier korrekt typisiert ist und durch den Defaultwert kein weiteres Problem darstellt.

2.2 Ontologien in der Informatik

Unter Ontologien versteht man in der Informatik eine Spezifikation einer Konzeptualisierung [3]. Dies bedeutet, dass die Entitäten eines Themenfeldes mit einer Vielzahl von Beziehungen verbunden werden, so dass ein Graph entsteht.

Hierbei unterscheidet man zwischen Toplevel- und Domain-Ontologien, wobei der Unterschied bei der Wahl des Themenfeldes liegt: Während sich Domain-Ontologien auf ein mehr oder weniger scharf umrissenes Themengebiet konzentrieren und versuchen dieses detailliert abzubilden, ist das Ziel von Toplevel-Ontologien möglichst viele Informationen aufzunehmen und sich hierbei auch ganze Domain-Ontologien zu eigen zu machen.

Eine besonders zu erwähnende Toplevel-Ontologie ist die „Sugested Upper Merged Ontology“ (SUMO). Hierbei handelt es sich um eine Ontologie die versucht Dinge möglichst vollständig zu axiomatisieren und sich hierbei auch selbst zu beschreiben. So sind etwa die verschiedenen Beziehungen zwischen verschiedenen Entitäten selbst wieder Entitäten. Zur Spezifikation von SUMO wird SUO-KIF, ein Lisp-Dialekt, verwendet. Erklärtes Ziel von SUMO ist es, dass die Eindeutigkeit der Beziehungen nicht von Namen oder Beschreibungen in normaler Sprache abhängt, sondern zur Gänze durch die logische Struktur der Netzwerkes beschrieben wird. Ein Aspekt der SUMO sehr mächtig macht ist die Benutzung von Prädikatenlogik höherer Ordnung welche auch die Beschreibung von Axiomen über Prädikaten ermöglicht. [9]

Ein Resultat dieser Selbstbeschreibung ist jedoch, dass es schwer ist, diese Ontologie Schritt für Schritt aufzubauen, da beispielsweise die Teilmenge aus `Entity`, `Relation` und `Subclass` kaum in dieser Reihenfolge erzeugt werden kann, da in Schritt 2 keine vollständige Abbildung existieren kann: Sind `Entity` und `Relation` bekannt, so kann nicht ausgedrückt werden, dass Relation eine Unterklasse von `Entity` ist. Sind `Entity` und `Subclass` bekannt, kann nicht formuliert werden, was eine Relation ist wodurch `Subclass` bedeutungslos wird. Sind `Relation` und `Subclass` bekannt, so fehlt die Wurzel der kompletten Hierarchie.

Ein anderer Ansatz wird von der „Web Ontology Language“ (OWL) verfolgt, einer formalisierten Sprache zur Definition eingener Ontologien. Im Gegensatz zu SUMO ist die Ausdruckstärke duch die Begrenzung auf Prädikatenlogik erster Ordnung sehr beschränkt. Zwar führt dies zu starken Einschränkungen mit Bezug auf die Funktionalität, erleichtert aber die Untersuchung beträchtlich. OWL-Ontologien werden typischerweise in XML gespeichert, einer im Web-Umfeld verbreiteten aber im Vergleich zu SUO-KIF erheblich aufwändiger zu parsenden Beschreibungssprache.

OWL untergliedert sich in drei Teilmengen mit aufsteigender Mächtigkeit: „OWL-Lite“, „OWL-DL“ und „OWL-Full“. OWL-Full ist hierbei unentscheidbar, während OWL-Lite für viele Zwecke zu eingeschränkt ist. In der Praxis wird daher primär OWL-DL benutzt dass sich als für viele Zwecke brauchbarer Kompromiss herausgestellt hat.

Zu den wesentlichen Unterschieden zwischen OWL und SUMO gehört die im Vergleich enorme Beschränktheit von OWL: Während SUMO nahezu beliebige Informationen in Prädikatenlogik höherer Ordnung darstellen kann, ist in OWL nur eine Teilmenge von Logik erster Ordnung erlaubt in der die Prädikate mit mehr als drei Werten nicht erlaubt sind. Die praktische Beschränktheit dieses Ansatzes wird zum Beispiel daran ersichtlich, dass es in OWL nur sehr aufwändig und unelegant möglich ist, auszudrücken, dass ein Individuum sich nach einer bestimmten Metrik zwischen zwei anderen befindet, während dies in SUMO sogar mit parametrisierbarer Metrik nahezu trivial ist. Auf der anderen Seite bedeutet diese Mächtigkeit aber auch, dass es erheblich aufwändiger ist, Probleminstanzen zu entscheiden und die Auswahl an verfügbaren Beweisern im Vergleich erheblich eingeschränkt ist (die meisten freien Beweiser arbeiten nur auf Prädikatenlogik erster Ordnung).

2.3 Aktueller Stand der Technik

Heutzutage werden Ontologien primär mit eigens hierfür geschriebenen Editoren erstellt, besonders bekannt ist in diesem Zusammenhang Protègè. [6] Was die Verwendung aus Programmiersprachen heraus angeht, existiert ein vergleichsweise breites Spektrum an Ansätzen, von denen im Folgenden ausgewählte näher beschrieben werden.

Zunächst wären hier jene Ansätze zu nennen, welche Entitäten einer Ontologie als Instanzen einer OOP-Klasse repräsentieren und dabei weniger an einer direkten Reproduktion der Ontologie interessiert sind, als vielmehr daran, ein programmierbares Werkzeug für die Bearbeitung einer Ontologie darzustellen. Diese Ansätze werden auch als indirekte Ansätze bezeichnet [10]. Zu nennen sind hier insbesondere die OWL-API [4] und Jena [5] aus dem Apache-Projekt.

Bei der OWL-API handelt es sich um eine Java-Bibliothek welche OWL-Trippel mit einem vergleichsweise niedrigen Abstraktionsgrad bietet und ein allgemeines Beweiser-Interface

bietet, so dass dieser ohne große Änderungen ausgetauscht werden kann. Die API bietet zum einen die Möglichkeit Anfragen zu stellen, als auch die Ontologie selbst zu ändern. Erwähnenswert ist noch, das die OWL-API von Protègè als Backend benutzt wird.

Apache Jena wählt einen in der Grundausrichtung vergleichbaren Ansatz, bietet aber auch höhere Abstraktionslevel. Die Grundidee von Jena ist es, Daten und Typen aus Ontologien als Instanzen von Unterklassen von `OntRessource` (z.B. `Ressource`) darzustellen und diese über allgemeine Methoden zu benutzen. Angenommen etwa, `base` sei eine Instanz von `OntModel` und enthalte eine Ontologie deren Basis-URL in `SOURCE` gespeichert ist und die eine Klasse `Paper` enthält, so kann ein neues Individuum das eine Instanz von `Paper` ist, mit folgendem Code erzeugt werden:

```
// Source: https://jena.apache.org/documentation/ontology/
OntClass paper = base.getOntClass( SOURCE + "#Paper" );
Individual p1 = base.createIndividual( SOURCE + "#paper1", paper );
```

Deutlich zu erkennen ist die fehlende Typisierung der Ontologieobjekte in der Java-Umgebung sowie der starke Fokus auf die Benutzung von Zeichenketten.

Darüber hinaus ist es mit Jena auch möglich einfache Anfragen in Java zu stellen (etwa um über die Klassen eines Individuums zu iterieren), allerdings ist die bevorzugte Technologie für komplexere Anfragen die Anfrage-Sprache SPARQL (SPARQL Protocol And RDF Query Language).

SPARQL stellt eine vom W3C standartisierte Sprache dar, die sich an SQL (Structured Query Language) anlehnt und erlaubt vergleichsweise komplexe Anfragen zu formulieren, die dann von einem Beweiser beantwortet werden.

Auf der anderen Seite gibt es jedoch auch Versuche die Daten von Ontologien in reguläre OOP-Sprachen zu übersetzen, was auch als direkter Ansatz bezeichnet wird [10] . Hierbei ergeben sich jedoch substantielle Probleme, da Ontologien sehr viel allgemeiner und Ausdrucksstärker als die meisten OOP-Sprachen sind. Dennoch finden sich in der Literatur einige interessante Ansätze.

ActiveRDF [8] umgeht die üblichen Beschränkungen dadurch, dass es statt wie sonst im Ontologien-Umfeld stark verbreitet nicht Java als OOP-Sprache benutzt, sondern stattdessen auf Ruby setzt, welches mit seinem dynamischen Typsystem, und der Möglichkeit Aufrufe von nicht existierenden Funktionen abzufangen und zu behandeln, viele der sonst üblichen Probleme umgeht. Einen substantiellen Nachteil hiervon stellt jedoch das dynamische Typsystem dar, welches viele logische Fehler, die sich in statisch typisierten Sprachen nicht oder nur schwer formulieren lassen, übersieht und die daraus resultierenden Probleme unter Umständen erst sehr spät oder gar nicht detektiert.

Auch der in „OWL vs. Object Oriented Programming“ [7] verfolgte Ansatz „SWCLOS“ verwendet mit Lisp und CLOS eine dynamisch typisierte Sprache für die Implementierung. Im Gegensatz zum hier verfolgten Ansatz konzentriert sich der dort beschriebene sich jedoch erheblich stärker auf die mögliche Typisierung von Programmen selbst und Anfragen bezüglich deren Konsistenz, als auf die semantischen Beziehungen zwischen dynamisch erzeugten Objekten.

Ein weiterer Ansatz, der versucht Ontologien und objektorientierte Programmierung zu vereinen, wird in „Integrating Object-Oriented and Ontological Representations: A Case

study in Java and OWL“ [10] beschrieben. Dort geht es effektiv darum, sowohl Eigenschaften direkter als auch indirekter Repräsentationen zu vereinigen. Im Unterschied zum hier beschriebenen Ansatz sind allerdings Entitäten und Objekte nach wie vor größtenteils getrennt und kommunizieren an den „Rändern“ miteinander. Der Vorteil dieses Ansatzes soll in erster Linie sein, dass so die Grenze zwischen diesen „Welten“ verschoben werden kann. Auch dieser Ansatz unterscheidet sich damit fundamental von dem in dieser Arbeit beschriebenen, der vielmehr alle Daten in beiden „Welten“ als direkt benutzbar und typisiert vorhält.

2.4 Einsatzgebiete

Ein zentrales Einsatzgebiet von Ontologien findet sich im Kontext des „Semantic Web“, einer Initiative von unter Anderem Tim Berners-Lee welche sich zum Ziel gesetzt hat zumindest Teile des WWW (World Wide Web) maschinenlesbar zu machen. Hierzu können beispielsweise menschenlesbare Texte im Html-Format so annotiert werden, dass ein Programm dazu in der Lage ist, die Daten auch in einer für OWL geeigneten Form zu extrahieren. Hierdurch könnte das im Web vorliegende Wissen in seiner Gesamtheit besser nutzbar gemacht werden.

Davon abgesehen gibt es jedoch auch interessante Versuche auf anderen Gebieten, etwa mit „Smart-API“ [2], dem Versuch, eine (Java-) API so zu annotieren, dass etwa Autovervollständigungswerkzeuge erheblich besser darin werden die Absichten der Benutzer einzuschätzen und nach Möglichkeit große Mengen an korrektem Quelltext automatisch generieren können.

Eine denkbare Fortsetzung dieses Ansatzes wäre eine entsprechend annotierte API stattdessen dazu zu benutzen um fortgeschrittene statische Codeanalyse in benutzenden Programmen auszuführen und so sehr viel komplexere Programmierfehler zu finden als solche die von aktuellen Werkzeugen erkannt werden. Gegenwärtig dürfte eines der Probleme für diesen Ansatz der potentiell große Aufwand sein, ein vollständiges Programm in einer Ontologie darzustellen, allerdings ist es denkbar, dass sich diese Problematik mit den in dieser Arbeit untersuchten Ansätzen in Verbindung mit Programmbibliotheken wie Libclang reduziert.

2.5 Abgrenzung der eigenen Arbeit

Der in dieser Arbeit untersuchte Ansatz grenzt sich von den oben beschriebenen Versuchen Ontologien und reguläre Programmiersprachen in Verbindung zu bringen dadurch ab, dass die Übersetzungsrichtung nicht, wie sonst üblich ist, die Ontologie in die Programmiersprache abzubilden, sondern die Programmiersprache in die Ontologie. Darüber hinaus wird die Nutzung der deklarativen Aspekte von Ontologien auch zur Laufzeit und in Verbindung mit klassischen Objekten ermöglicht. Hierdurch werden die sonst üblichen Probleme einer solchen Übersetzung dadurch umgangen, dass eine derartige Übersetzung nicht, wie etwa in [8] beschrieben, zwingend verlustbehaftet ist:

Tabelle 2.1: Probleme bei der Übersetzung zwischen Ontologien und OOP: Die Ontologien sind sehr viel mächtiger, entsprechend ist primär eine Übersetzung zu ihnen möglich, während die umgekehrte Richtung Schwierigkeiten mit sich bringt.

Ontologie → OOP	OOP → Ontologie
Die Abbildung der dynamisch aufgebauten Objekteinträge einer Ontologie in ein statisches Typsystem ist kaum auf angenehme Art und Weise zu bewerkstelligen.	Die Abbildung der Objekte eines statischen Typsystems in ein dynamisches welches eine Obermenge an Zuständen erlaubt, ist problemlos möglich
In Ontologien können beliebige Objekte miteinander verbunden sein, was sich praktisch kaum auf normale Attribute in OOP- Sprachen übertragen lässt	Da Attribut-Beziehungen äußerst simpel sind, ist es ein leichtes, sie in Ontologien darzustellen.
Die komplexen, potentiell zyklischen Beziehungen zwischen Klassen einer Ontologie lassen sich nicht auf azyklischen Vererbungsgraphen abbilden. (außerhalb von C++ kommt das häufige Fehlen von Mehrfachvererbung erschwerend hinzu)	Die gerichteten, azyklischen Graphen einer Vererbungshierarchie lassen sich ohne Probleme auf die allgemeinen Graphen einer Ontologie abbilden.
Die Evolution des Schemas einer Ontologie ist ein zentraler Mechanismus in ihrer Verwendung und bringt entsprechend große Probleme bei der Übersetzung in eine statische Sprache, bei der die Definition einer Klasse zur Laufzeit konstant ist	Die Abbildung der konstanten Klassenschemata in eine Ontologie die *auch* dynamische Schemata erlauben würde ist trivial.

ActiveRDF umgeht die genannten Probleme dadurch, dass es als OOP-Sprache Ruby benutzt, welches dynamisch typisiert ist und ein hohes Maß an Flexibilität bietet, um Klassen auch zur Laufzeit noch zu ändern.

Während der dort untersuchte Ansatz sehr interessant ist und insbesondere in der Zielsetzung auch durchaus Parallelen zu dem hier verfolgten erkennbar sind, gibt es jedoch auch fundamentale Unterschiede:

- Die Wahl der Programmiersprachen impliziert einen großen Unterschied im benutzten Typsystem: Während Ruby dynamisch ist, handelt es sich bei C++ klar um einen Vertreter statischer Sprachen.
- Die Übersetzungsrichtung unterscheidet sich: ActiveRDF sieht nicht vor native Ruby-Typen in Ontologien einzubetten, während dies ein zentrales Ziel des hier untersuchten Ansatzes ist.
- Active-RDF benutzt Proxyobjekte, die im Prinzip alle den selben Typen haben und bei allen Methodenaufrufen unbekannter Methoden versuchen eine Relation in der

Ontologie zu finden und die Anfrage hierüber zu beantworten. Auf der anderen Seite benutzt unser Ansatz reguläre C++-Objekte, die auch außerhalb der Ontologie nützlich sein können und benötigt dafür explizitere Anfragen an den Beweiser.

3 Untersuchte Ansätze

3.1 Allgemeine Techniken

Für die Mehrheit der im folgenden vorgestellten Techniken ist es nötig, dass alle Benutzten Typen eine gemeinsame Basisklasse haben, diese wird im Folgenden immer `entity` sein. Während dies zwar keine all zu große Einschränkung für neu definierte Typen ist, handelt es sich hierbei um eine große Schwierigkeit bei der Benutzung existierender Typen und um eine faktisch unerfüllbare Anforderung für existierende Typen in fremden Bibliotheken oder eingebaute Typen wie `int` oder `double`.

Die hierfür entworfene Lösung ist die Einführung eines kleinen Klassentemplates `e`, welches existierende Typen `T` als `e<T>` wrappt und selbst von `entity` erbt. Eine Implementierung kann wie folgt aussehen:

```
template<typename T>
class e: public entity {
public:
    template<typename...Args>
    e(Args&&... args): value{std::forward<Args>(args)...} {}

    T& operator*() {return value;}
    const T& operator*() const {return value;}
    T* operator->() {return &value;}
    const T* operator->() const {return &value;}
private:
    T value;
};
```

Eine mögliche Erweiterung wäre das Hinzufügen einer impliziten Typkonvertierung nach `T&` und/oder `const T&`, wobei hier jedoch zu beachten ist, dass eine solche Konvertierung nicht für Methoden funktioniert und auch potentiell im Zusammenhang mit Templates zu Problemen führen kann.

3.2 Repräsentation der Ontologie

Die exakten Implementierungen der Ontologien variieren naturgemäß stark mit der grundlegenden Architektur des verfolgten Ansatzes, generell haben sich aber gemeinsame Muster und Optionen herauskristallisiert.

Prinzipiell muss es eine gemeinsame Ontologie-Klasse geben, die Relationen, Axiome und Individuen kennt und dieses Wissen verwaltet. Als allgemein sinnvoll hat sich hier erwiesen,

dass von Individuen verlangt wird, dass ihr Typ von `entity` erbt, was der Ontologie ermöglicht, alle ihre Individuen über Zeiger auf `entity` zu speichern. Um Individuen eindeutig identifizieren zu können muss nun zusätzlich jedes Individuum einen eindeutigen Bezeichner haben, wofür sich die Verwendung von `std::size_t` anbietet. Dieser Bezeichner wird nun (in der Regel gemeinsam mit einem Zeiger auf die zugehörige Ontologie) in `entity` gespeichert.

Indem nun für Moves geeignete Konstruktoren und Zuweisungsoperatoren sowie Destruktoren für `entity` definiert werden, ist es möglich Individuen zuverlässig zu verfolgen und sie am Ende ihrer Lebenszeit auch zuverlässig wieder zu entfernen.

3.3 Anbindung an Beweiser

Im Verlauf der Entwicklung kamen zwei Beweiser zum Einsatz die beide nach dem selben Prinzip eingebunden wurden: SPASS [11] und Z3 [1]. Bei beiden handelt es sich um Beweiser für Prädikatenlogik erster Ordnung, was eine Abbildung von SUMO leider unmöglich macht, aber dennoch sehr komplexe Anfragen erlaubt, die über das hinausgehen, was OWL bietet. Bei SPASS handelt es sich um eine Entwicklung des Max Planck Instituts für Informatik, während Z3 von Microsoft Research stammt. Beide Programme sind unter einer freien Lizenz verfügbar.

Aufgrund positiver Erfahrungen mit derartigen Ansätzen in früheren Projekten wurde für die Kommunikation eine Multiprozessarchitektur gewählt, welche sicherstellt, dass kein Prozess den Anderen auf unerwünschte Art stören kann. Konkret sieht die Implementierung so aus, dass bei einer Anfrage der Beweiser als neuer Kindprozess gestartet wird, dessen Standardein- und -Ausgabe mit C++-Streams des Elternprozesses verbunden sind. Über diese Streams wird nun das Problem in textueller Form übertragen und letztlich die Ausgabe nach einer Antwort durchsucht.

Im Falle von Z3 als Beweiser wird jedoch statt bei jeder Anfrage einen neuen Prozess zu starten, pro Thread nur ein einziger erzeugt der dann wiederverwendet wird. Am sonstigen Prinzip ändert sich jedoch nichts.

3.4 Dynamische Funktionsausführung

Eine der initialen Ideen für mögliche Anwendungen von SOOP war es, komplexe Operationen durch die Kombination existierender C++-Funktionen die vom Reasoner ausgewählt wurden dynamisch zu erzeugen. Auch wenn dieser Ansatz letzten Endes im Rahmen dieser Arbeit nicht weiterverfolgt wurde, wäre dies sicher ein interessantes Gebiet für weitere Forschung.

Die prinzipielle Problematik Funktionen dynamisch auszuwählen und zu verketten ist die statische Typisierung der selben: Zwar wäre es möglich zu verlangen das jede Funktion etwa eine `std::initializer_list<entity*>` als einziges Argument entgegennimmt, allerdings wäre dies eine enorme Einschränkung auf der Seite der Definition und würde massiv die eigentlich gewünschte Typsicherheit untergraben. Darüber hinaus wäre eine Verwendung existierender Funktionen faktisch unmöglich. Insbesondere die erste Einschränkung ist für C++-Programmierer klar inakzeptabel, da sie massiv unidiomatischen Code erzwingt.

Um diese Probleme zu umgehen, beziehungsweise zu reduzieren, wurde daher ein Ansatz entworfen der Typauslöschung benutzt um zwar einerseits eine dynamische Anzahl und dynamische Typen für die Funktionsargumente zu erlauben, andererseits aber weder an der Stelle der Funktionsdefinition, noch bei regulärer Benutzung auf Typsicherheit verzichten zu müssen.

Der Einfachheit halber werden wir zunächst folgende Annahmen treffen:

- Alle Funktionen erhalten nur konstante Referenzen auf Instanzen von Klassen welche von `entity` erben, als Argumente.
- Alle Funktionen geben eine Instanz einer von Entity erbenden Klasse zurück.
- Alle Funktionen „`Self`“ sind Funktionsobjekte, die von `relation<Self, Returntype, Argumenttypes>` erben.

Zunächst seien folgende Definitionen gegeben:

```
class entity {
public:
    virtual ~entity() = default;
};

using entity_ref_list = std::vector<const entity*>;
```

Die Klasse `entity` repräsentiert hierbei die Basis der fraglichen Ontologie und nimmt eine ähnliche Rolle wie Javas `Object` ein. Dies ist unter anderem nötig um zur Laufzeit zu entscheiden, ob ein Object eine Instanz eines bestimmten Typen ist.

Als nächstes wird ein Funktionstemplate `call` definiert, welches ein Funktionsobjekt `f` und eine `entity_ref_list` mit seinen Argumenten erhält, sowie, per Templateargument, die Liste der benötigten Typen und einen Parameter-pack mit den benötigten Indizes um die Argumente zu erhalten:

```
template <typename Fun, typename... Args, std::size_t... Indices>
auto call(const Fun& f, const entity_ref_list& args,
          std::index_sequence<Indices...>) {
    return f(dynamic_cast<const Args&>(*args.at(Indices))...);
}
```

Dank des Parameter Packs ist es möglich die Funktion direkt aufzurufen, da so der dynamische Zugriff auf n Argumente durch einen statischen ersetzt wird. Mittels des `dynamic_cast` ist es ferner möglich, die Typen wieder in eine für den Funktionsaufruf geeignete Form zu bringen.

Zuletzt muss eine Möglichkeit geschaffen werden, alle Funktionen unabhängig von ihrer Signatur über den selben Typen darzustellen. Hierzu wird wieder Vererbung benutzt. Zunächst wird eine allgemeine Basisklasse definiert welche den (rein virtuellen) Aufruf-Operator dahingehend überlädt, dass er eine `entity_ref_list` als Argument entgegen nimmt und einen `std::unique_ptr<entity>` zurück gibt:

```
class base_relation: public entity {
public:
    virtual std::unique_ptr<entity> operator()(
```

```
            const entity_ref_list&) const = 0;
};
```

Zur implementierung wird nun ein Klassentemplate, `relation`, welches von `base_relation` erbt, aber via CRTP („Curiously Recurring Template Pattern“: Eine Technik um nicht-virtuelle Methoden aus einer Basisklasse aufzurufen) nähere Details über die eigentlich aufzurufende Funktion kennt, implementiert:

```
template <typename Self, typename Ret, typename... Args>
class relation : public base_relation {
public:
    std::unique_ptr<entity> operator()(
            const entity_ref_list& args) const override {
        auto ret = call<Self, Args...>(
                static_cast<const Self&>(*this), args,
                std::make_index_sequence<sizeof...(Args)>{});
        return std::make_unique<Ret>(std::move(ret));
    }
};
```

Im Prinzip besteht die Aufgabe dieses Funktors nur daraus, `call` aufzurufen und das (Stack-)Ergebnis in einem `std::unique_ptr<entity>` zu speichern, so dass es für eine weitere Verwendung geeignet ist. Hierfür castet sich zunächst das Funktionsobjekt selbst zu seinem eigentlichen Typ und erzeugt anschließend geeignete `std::index_sequence`, die dann wie oben beschrieben von `call` benutzt wird.

Im folgenden sein nun Beispielhaft demonstriert, wie diese Technik dazu benutzt werden kann um einen Objekt eines Typs `From` zu einem des Typs `To` zu transformieren, indem eine von einem Reasoner erdachte Abfolge von Aufrufen diverser Funktionen zu verwenden.

Zunächst seinen folgende Typen gegeben:

```
struct operation {
    std::vector<std::size_t> inputs;
    std::size_t output;
    const base_relation* rel;
};

using op_list = std::vector<operation>;
```

Eine `operation` repräsentiert hierbei einen Rechenschritt in der Konvertierung. Die in `inputs` und `output` gespeicherten Werte repräsentieren hierbei den Index an dem die Funktionsargumente in einem `std::vector` temporärer Werte gespeichert sind, beziehungsweise die Ausgabe gespeichert wird (im folgenden Codeauszug: `data`). Hierbei muss für alle Indizes `i` in `input` gelten: `output > i`. Bei `op_list` handelt es sich ferner um eine komplette Abfolge von Rechenschritten die benötigt werden um letztlich an das Ergebnis zu kommen (gewissermaßen das vollständige Programm). Hierbei muss für jede `operation` o gelten, dass jeder der Werte in ihren `inputs` entweder 0 ist, oder der Wert des `output` einer `operation` o2 die vor o in der Liste steht ist.

Zu guter Letzt kann nun die eigentliche Transformationsfunktion implementiert werden:

```
template <typename To, typename From>
To transform_to(From from, const op_list& ops) {
    std::vector<std::unique_ptr<entity>> data(ops.back().output + 1u);
    data.front() = std::make_unique<From>(std::move(from));
    for (const auto& op : ops) {
        entity_ref_list args{};
        for (auto index : op.inputs) {
            args.push_back(data.at(index).get());
        }
        data[op.output] = (*op.rel)(args);
    }
    return std::move(dynamic_cast<To&>(*data.back()));
}
```

Hierbei wird zunächst ein `std::vector` namens `data` angelegt der die Zwischenergebnisse speichert. Da der höchste Index gemäß der oben definierten Anforderungen der `output` der letzten `operation` ist, kann dieser Wert + 1 als Größe von `data` angenommen werden.

Zu Beginn enthält `data` noch keine Werte (aka nur Nullzeiger); um nun die erste Operation ausführen zu können, wird der Ausgangswert vom Typ `From` am Index 0 gespeichert. Dann wird für jede `operation` zunächst eine Argumentliste von Typ `entity_ref_list` erstellt indem die Parameter aus dem `data`-Vector ausgelesen werden und letztlich das Ergebnis wieder in `data` gespeichert. Nachdem die letzte `operation` ausgeführt wurde, wird nun der letzte Wert in `data` zum Zieltyp `To` gecasted und zurückgegeben.

Eine beispielhafte Anwendung könnte so aussehen: Gegeben seien zwei Typen für zweidimensionale Vektoren `v1` und `v2`, sowie je ein Typ für x- und y-Koordinaten `x` und `y`. Ferner existieren Funktionen um `x` und `y` von `v1` zu erhalten, sowie ein Konstructor `from_xy` der eine Instanz von `v2` aus Instanzen von `x` und `y` erzeugen kann.

Auf die Frage wie man eine Instanz von `v1` zu einer von `v2` konvertieren kann, gibt ein Reasoner nun die offensichtliche Antwort. Mit den obigen Techniken ist es nun zur Laufzeit möglich diese Funktionen zu benutzen um die vorgeschlagene Abfolge von Funktionsaufrufen zu verwenden:

```
const v1 in = { ... };
// get this list from reasoner:
const op_list ops = { {{0}, 1, x}, {{0}, 2, y}, {{1,2}, 3, from_xy} };
const v2 out = transform_to<v2>(in, ops);
```

Zuletzt zu den ursprünglich genannten Einschränkungen: Die Limitierung auf Typen die von `entity` erben wird durch den Einsatz des `e`-Templates weniger gravierend. Um reguläre Funktionen zu benutzen lässt sich darüber hinaus ebenfalls ein Wrapper schreiben, der über die Destrukturierung von Funktionszeigern als Templateargumenten alle Informationen erhält die für die Benutzung von `relation` nötig sind.

3.5 Statisches LISP

Ein zunächst untersuchter aber aus den später genannten Gründen letztlich verworfener Ansatz für die allgemeine Implementierung war, weite Teile der Axiomatisierung bereits zur Übersetzungszeit mittels Templates zu formulieren.

Hierfür wurde ein `problem`-Klassentemplate verfasst, welches drei Argumente verlangte: Eine Liste der benutzten Typen, eine Liste der verwendeten Prädikate und eine Liste von Axiomen. Diese Listen waren wiederum variadische Klassentemplates welche die Werte enthielten. Um die korrekte Instanziierung von `problem` zu garantieren wurde nun ein allgemeines Template dieses Namens deklariert und eine geeignete teilweise Spezialisierung definiert. Hierdurch würden viele falsche Aufrufe dazu führen, dass ein früher (und damit klarer) Fehler wegen der Benutzung einer nicht definierten Templatespezialisierung ausgegeben wird.

Vereinfacht sah das `problem`-Template damit so aus:

```
template<typename Types, typename Rels, typename Axioms>
class problem;

template<typename...Types, typename...Rels, typename...Axioms>
class problem<
    functions<Types...>,
    predicates<Rels...>,
    formulae<Axioms...>
> { /* actual implementation */ };
```

Die Namen `functions`, `predicates` und `formulae` sind hierbei ein Resultat des zur Entwicklungszeit benutzten Beweisers SPASS, dessen Eingabe diese Namen benutzt. Wäre der Entwurf weiterentwickelt wurden, so wären hier vermutlich andere Namen zum Einsatz gekommen. Zu jedem dieser Listentemplates gab es ferner ein Template das eine einzelne Instanz enthielt (nur diese „Singulartemplates“ durften in den Listentemplates gespeichert werden), namentlich `function`, `predicate` und `formula`. Alle Instanzen dieser Templates besaßen eine statische Methode `to_string`, welche eine für den Beweiser geeignete textuelle Repräsentation der Instanz lieferten.

Die Grundidee des Ansatzes war nun zunächst alle Individuen, Prädikate und Axiome der Ontologie über Typen zu repräsentieren. Hierbei wurden Individuen als normale Typen und Prädikate als Klassentemplates repräsentiert, deren Typargumente den Prädikatargumenten entsprach. Durch die Kombination von Individuen und Prädikaten ließen sich nun leicht komplexe Formeln darstellen, die letztlich als Axiome dienten. So wird im Folgenden etwa gezeigt, wie eine Relation `less` definiert werden kann, welche Typen vergleicht und wie sich diese als transitiv annotieren lässt.

Zunächst werden dafür als Individuen geeignete Typen definiert:

```
struct t1 {
    static std::string to_string() { return "v1"; }
    // further functionality skipped for simplicity
};
```

Analog zu `t1` können weitere Typen `t2`, `t3` und `tn` definiert werden.

Als nächstes kann `less` definiert werden, was wie folgt aussieht:

```
template<typename L, typename R>
struct less {
    static std::string name() { return "less"; }
    static std::string to_string() {
        return name()+"("+L::to_string()+", "+R::to_string()+")";
    }
    static unsigned rank() { return 2; }
};
```

Zuletzt kann nun ein Axiom zusammengebaut werden, welches die Transitivität von `less` formal definiert:

```
using example_formula = formula<forall<vars<v1,v2,v3>,
    implies<
        and_<less<v1, v2>, less<v2,v3>>,
        less<v1,v3>
    >>>;
```

Gut zu sehen ist hier auch, warum dieser Ansatz „statisches LISP“ getauft wurde: Statt runder Klammern werden spitze benutzt und Funktionen stehen außerhalb, nicht innerhalb der Argumentlisten, aber von diesen Oberflächlichkeiten abgesehen, besteht eine überraschende semantische Ähnlichkeit.

Um nun Anfragen an den Beweiser zu stellen, werden ebenfalls Instanzen von `formula` verwendet und an ein Methodentemplate übergeben:

```
problem<...> p;
// assuming that e1 and e2 are types known to
// the ontology and ordered with regards to less:
std::cout << p.request<less<e1, e2>>();
```

Nicht nur da Fehlermeldungen die eine Instanziierung von `problem` beinhalten sehr schnell sehr groß und komplex werden, kann es sehr sinnvoll sein, weite Teile der API von `problem` in eine Basisklasse `basic_problem` auszulagern, welche sich bei Anfragen die statisch gesetzten Typen und Axiome nur noch aus der erbenden Template-Instanz über virtuelle Methoden beschafft. Hierbei deutet sich allerdings bereits an, dass die statische Formulierung von Axiomen nur einen begrenzten Nutzen hat.

Tatsächlich spielt dies auch schon auf den Hauptnachteil dieses Ansatzes (wenn in Reinform) an: Seine Beschränktheit auf Axiome welche zur Übersetzungszeit bekannt sein müssen. Zwar sind hierfür durchaus Anwendungen denkbar (etwa zur Verifikation statischer Annahmen im Typsystem), allerdings werden diese in der Praxis dadurch eingeschränkt, dass es nicht sinnvoll möglich ist, einen externen Beweiser während des Übersetzungsvorgangs anzusprechen und kein Beweiser bekannt ist, der in C++-templates verfasst wurde und sich somit direkt integrieren ließe.

Ohne Erweiterungen die dynamische Aspekte einführen stehen die, durch die genannten Probleme stark beeinträchtigten, Vorteile in einem eher ungünstigen Verhältnis zu den Nachteilen. Eine Weiterverfolgung des rein statischen Ansatzes erscheint daher höchstens

mit nativer Integration eines Beweisers in den Compiler sinnvoll. Als reine Bibliothekslösung ist von ihm jedoch abzuraten.

Durch die zusätzliche Unterstützung dynamischer Aspekte ändert sich dieses Fazit allerdings zumindest teilweise. Der wichtigste Aspekt hierbei ist naturgemäß die Betrachtung von regulären Objekten, in der Regel also Instanzen der der Ontologie bekannten Klassen. Die Grundlage hierfür ist die bereits zu Beginn beschriebene Basisklasse `entity` mit dem Wrappertemplate `e`. Die Aufgabe der Ontologie ist es nun sich sämtliche Entitäten zu merken (etwa über einen `std::vector<const entity*>`), während diese bei ihrem Ableben (also in ihrem Destruktor) der Ontologie mitteilen müssen, dass sie nicht länger existieren. Ohne Relationen ließen sich allerdings höchstens Instanzbeziehungen zwischen Entitäten und Klassen via RTTI („Runtime Type Information“) herleiten, weswegen zusätzlich dynamische Relationen benötigt werden. Hierdurch wird dann zwar der komplette Ansatz erheblich dynamischer, führt damit jedoch auch die Grundidee in gewisser Weise ad Absurdum.

Zusammenfassend lässt sich hier also sagen, dass die Praxisrelevanz eher gering ist, auch wenn bei der Untersuchung einzelne Techniken entwickelt wurden, die für die finale Lösung nützlich waren und entsprechend dort beschrieben werden.

3.6 Zeichenkettenbasiert

In mancher Hinsicht der exakt gegenteilige Ansatz war die starke Fokussierung auf reine Zeichenketten: Die Ontologieklasse wurde hier praktisch ausschließlich dazu benutzt um Listen von Zeichenketten zu verwalten welche dann an den Beweiser weitergeleitet wurden.

Das Problem bei diesem Ansatz ist zunächst, dass komplexe Ausdrücke in ihm als Zeichenketten dargestellt werden, aber die Entitäten in einem Programm stark typisiert vorliegen. Um dem Benutzer die komplizierte Erzeugung eines geeigneten Funktionstemplates zu ersparen werden daher Werkzeuge angeboten, die diese Erzeugung vergleichsweise einfach gestalten:

Zunächst wird ein allgemeines Klassentemplate `binder` definiert welches den Call-Operator mit einem variadischen Methodentemplate so überlädt, dass ein Aufruf mit geeigneten Argumenten letztlich eine Formel als `std::string` produziert. Das Klassentemplate `binder` erhält zwei Argumente, wovon das erste der Rang des Prädikats ist (entsprechend wird auch bei allen Aufrufen geprüft, ob die Zahl der Argumente diesem Wert entspricht) und optional eine Typliste die, wenn angegeben darüber hinaus eine stärkere Typisierung des Prädikats ermöglicht indem sie mittels `static_assert` sicherstellt, dass alle Argumente kompatibel zu den in der Typliste angegebenen sind.

Um nun die Formel zusammenzubauen speichert jede Instanz von `binder` den Namen des Prädikats welches sie gerade repräsentiert als `std::string`, sowie dessen ID als `std::size_t` und bietet Funktionen an die diese Wert auslesen. Um eine Instanz von `binder` zu erzeugen wird pro Prädikat ein Hilfstyp benötigt, welcher über statische Methoden den Namen und die ID des Prädikats mitteilt. Wird eine Instanz eines solchen Hilfstyps nun an den Konstruktor eines `binder` übergeben, wird dieser diese Werte auslesen und sich hiervon initialisieren.

Die Instanz des `binder` kann nun ähnlich einer herkömmlichen Funktion benutzt werden um hiermit fast beliebig komplexe Formeln zu erzeugen.

```
template <std::size_t Rank,
          typename TypeRequirements = allowed_types_list<void>>
```

```
class binder {
public:
    static_assert(Rank > 0, "");
    using required_types = TypeRequirements;

    template <typename F>
    binder(F f);

    template <typename... Args>
    std::string operator()(const Args&... args) const;

    const std::string& name() const { return m_name; }
    std::size_t id() const { return m_id; }
    constexpr static std::size_t rank() { return Rank; }

private:
    static const std::string& to_string(const std::string& s);
    static std::string to_string(const entity& e);
    std::size_t m_id;
    std::string m_name;
};
```

Die `to_string`-Methoden werden im Folgenden behandelt, zunächst sei jedoch eine beispielhafte Verwendung gezeigt:

```
struct foo_t {
    std::string name() {return "foo";}
    std::string id() {return 23;} // simplified
};
struct bar_t {}; // defined similar to foo_t

// foo takes two arguments:
const auto foo = binder<2>{foo_t{}};
// foo takes one arguments:
const auto bar = binder<1>{bar_t{}};

int main() {
    e<int> i{...}; // assume id = 1
    std::cout << foo(bar(i), bar("y"));
    // prints: foo(bar(o_1),(bar(y))
}
```

Um sowohl Objekte als auch komplexe Ausdrücke (die ja als `std::string` übergeben werden) angemessen auszugeben werden die statischen `to_string()`-Methoden von `binder` verwendet: Alle Argumente werden zur Konvertierung an diese übergeben, wodurch die Auflösung der Funktionsüberladung elegant die Benutzung von komplexeren Werkzeugen wie `enable_if` vermeidet. Konkret gibt die Version von `to_string` welche einen `std::string` erhält ihr Argument unverändert zurück, während die andere Version (welche eine Entität

erhält) sich deren Namen durch den Aufruf der Methoden `entity::name()` verschafft und diesen zurück gibt. Besagte `entity::name()` is wiederum so definiert, dass sie `"o_" + to_string(id())` zurück, gibt im Falle etwa der id 42 „`o_42`“.

Der große Vorteil dieses Ansatzes ist, dass die Implementierung *vergleichsweise* simpel bleiben kann (die Aufgabe der Ontologie bestand im Kern nur daraus Zeichenketten zu verwalten) und Prädikate höherer Ordnung zu einem gewissen Grad mit einfachen Funktionen emulieren ließen:

```
// The creation of a reusable higher-order-statements
// with this approach is trivial:
inline std::string transitive(const std::string& name) {
    return "forall([x,y,z], implies(and(" + name
        + "(x,y), " + name+"(y, z))," + name + "(x,z)))";
};
```

Dieses Beispiel zeigt, dass durch die Möglichkeit beliebige Zeichenketten als Axiome zu benutzen auch die Erstellung komplexerer Axiome vergleichsweise einfach möglich wurde. Hier wird eine Funktion definiert die dem Beweiser mitteilt, dass ein Prädikat transitiv ist. Während dies zwar auch in Prädikatenlogik erster Ordnung ausgedrückt werden kann, muss dies dort für jedes Prädikat einzeln geschehen; hier genügt eine einmalige Formulierung.

Zu den Nachteilen dieses Ansatzes gehört im Umkehrschluss, dass die semantische Benutzung der Axiome in C++ selbst relativ stark eingeschränkt ist, da hier ja nur Zeichenketten vorliegen, die entsprechend geparst werden müssten, was jedoch wieder alle Vorteile zunichte machen würde. Das große Problem an dieser Einschränkung ist, dass tiefe Kopien von Axiomen so praktisch unmöglich werden (zwar ist es möglich zwei Entitäten als gleich zu deklarieren, aber jede Änderung an der einen wird somit auch die Andere ändern, was eine hochgradig unidiomatische Implementierung von Kopieroperationen darstellen würde. Umgekehrt kann jedoch auch die Verwaltung der Axiome nicht entfernt werden, da Entitäten ja prinzipiell gelöscht werden können und die Ontologie als Reaktion hierauf auch alle verbundenen Axiome löschen muss.

Zuletzt stellt die eingeschränkte Typisierung ein allgemeines Problem dar, welches ja eigentlich durch die Verwendung von C++ vermieden werden soll.

Abschließend lässt sich sagen, dass einige der im Rahmen dieser Arbeit entwickelten Techniken vielversprechend erscheinen, auch wenn sie letztlich nicht verwendet wurden. Ferner behält der im folgenden Kapitel vorgestellte letztlich verwendete Ansatz auch einige Ideen die nur wenig gegenüber den hier vorgestellten verändert wurden.

4 SOOP

Dieses Kapitel wird den letztlich verwirklichten Ansatz beschreiben. Hierfür wird zunächst ein grober Überblick über die Gesamtarchitektur gegeben, gefolgt von einer detaillierteren Beschreibung einzelner Komponenten. Zuletzt wird schließlich auf einzelne Techniken in der Implementierung und eingebaute Prädikate eingegangen.

Die Abkürzung „SOOP“ steht für „**S**emantische **O**bjekt**O**rientierte **P**rogrammierung“ und wird von der Implementierung auch als Namensraum verwendet.

Für das bessere Verständnis des Kapitels sind im folgenden einige häufiger benutzte Begriffe definiert:

Atom, Individuum Ein konkretes, prädikatenlogisches Objekt über das Aussagen gemacht werden können.

Axiom Hier eine prädikatenlogische Aussage die als wahr definiert wird.

Entität Ein C++-Objekt welches ein Individuum repräsentiert oder, im prädikatenlogischen Kontext, generell ein Individuum.

Klasse Eine Definition des Aufbaus von C++-Objekten.

Objekt Eine Instanz einer C++-Klasse oder eine prädikatenlogische Entität.

Prädikat Ein benanntes Tupel das Individuen und Variablen in Beziehung setzt.

Quantor Allquantor($\forall$) und Existenzquantor($\exists$). Ermöglichen prädikatenlogische Aussagen über Existenz und Allgemeingültigkeit.

Variable Ein an einen Quantor gebundener lokaler Bezeichner der für alle Individuen stehen kann.

4.1 Grobaufbau

SOOP besteht im Wesentlichen aus vier Komponenten: Entitäten, Formeln mit Prädikaten, Ontologien und einem Beweiser.

Entitäten stellen hierbei die Atome der Ontologie dar und werden in C++ durch Typen und Instanzen der von `soop::entity` erbenden Klassen repräsentiert.

Formeln werden durch die Verbindung von Prädikaten mit Entitäten gebildet. Im Kontext von SOOP muss ein Prädikat von `soop::is_predicate` erben und zwei spezielle Methoden implementieren (dazu später mehr).

Unter einer Ontologie wird im Kontext von SOOP eine Instanz von `soop::ontology` verstanden. Es handelt sich dabei um eine Klasse welche Entitäten, Prädikate und aus

diesen zusammengesetzte Formeln speichert und eine Schnittstelle bietet über die komplexe Anfragen an den Beweiser gestellt werden können.

Die Schnittstelle für den Beweiser besteht im Grunde aus zwei Funktionen `try_proof` und `request_entities` die beide eine für den Beweiser (in diesem Fall Z3) geeignete Beschreibung der Ontologie als Zeichenkette entgegennehmen und im Falle `request_entities` eine Liste von Variablennamen deren Wert zu erfragen ist. Erstere Funktion beantwortet daraufhin über eine Anfrage an den Beweiser ob die Ontologie widerspruchsfrei ist während Zweitere eine Liste von Ids von Entitäten zurückliefert, welche als Variablenbelegung geeignet wären.

Die Grundidee ist nun eine Ontologie `o` zu erzeugen, Entitäten und Formeln (Axiome) zu dieser hinzuzufügen und anschließend Anzufragen ob das jeweilige Modell konsistent ist, beziehungsweise wie ein konsistentes Modell aussehen könnte.

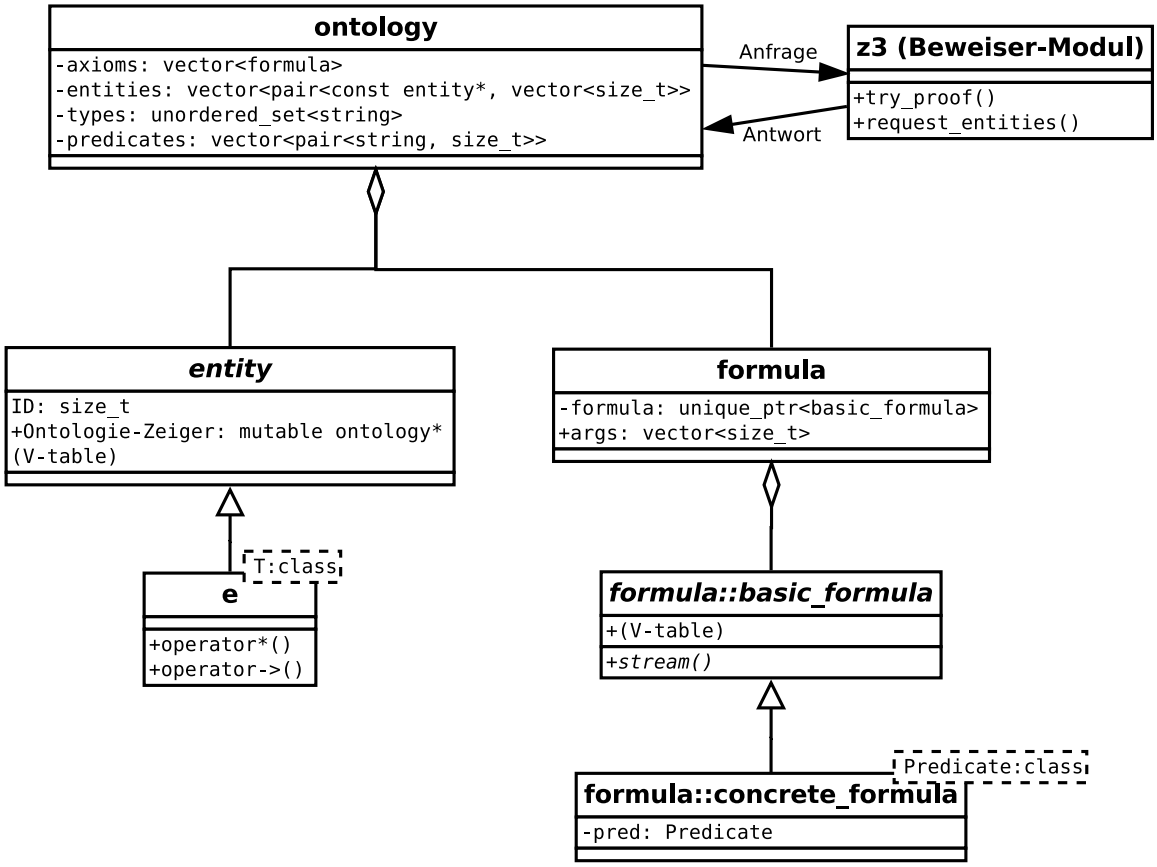

Abbildung 4.1: Überblick über die SOOP-Gesamtarchitektur.
Nicht dargestellt ist die Architektur der Prädikate, welche in `formula::basic_formula` gespeichert werden.

4.2 Architektur der Entitäten

Die Definition der allgemeinen Basisklasse `entity` sieht wie folgt aus:

```
class entity {
public:
    entity(ontology& o, const std::type_info& t);
    entity(std::nullptr_t);

    entity(entity&&) noexcept;
    entity& operator=(entity&&) noexcept;

    entity(const entity&) = delete;
    entity& operator=(const entity&) = delete;

    virtual ~entity();

    std::string name() const { return "o_" + std::to_string(m_id); }
    () const { return "o_" + std::to_string(m_id); }
    std::size_t id() const { return m_id; }
private:
    friend class ontology;
    mutable ontology* m_ontology = nullptr;
    std::size_t m_id = 0;
};
```

Entitäten *können*, müssen aber nicht Teil einer Ontologie sein. Was hier der Fall ist wird durch den `m_ontology`-Zeiger repräsentiert, der entweder auf die Ontologie zeigt zu der die Entität gehört oder ein `nullptr` ist, wenn es keine solche gibt (= freie Entität). Der Zeiger ist `mutable` um auch bei semantisch konstanten Objekten eine Verschiebung der Ontologie zu ermöglichen; die entsprechenden Operationen der `ontology`-Klasse korrigieren am Ende den Zeiger.

Da der primäre Einsatzzweck von Entitäten der Einsatz in einer Ontologie ist, wird für eine freie Entität eine explizite Entscheidung durch das Übergeben eines `nullptr` verlangt. Um eine gebundene (= nicht freie) Entität zu erzeugen wird wiederum eine Referenz auf die Ontologie benötigt und, da alle SOOP-Ontologien den Typ aller Laufzeitentitäten (= nicht Typentitäten), speichern auch eine `std::type_info` welche über den eigentlichen Typ der Entität informiert. Diese darf beliebige der Ontologie bekannte Basisklassen des eigentlichen Typs repräsentieren. Wird dieser Konstruktor verwendet, wird sich die neue Entität automatisch in der gewählten Ontologie registrieren.

Ein Vorteil hiervon ist etwa, dass der Ontologie hierdurch nicht beliebig komplexe Implementierungsdetails mitgeteilt werden müssen: Angenommen etwa, `matrix` sei eine rein abstrakte Basisklasse die von `entity` erbt und von `dense_matrix` und `sparse_matrix` implementiert wird. Hierbei handelt es sich um eine wichtige Platzoptimierung für Rechnungen in C++ mit denen die Ontologie aber ohnehin nichts anfangen kann. In diesem Fall ist es nun möglich dem `entity`-Konstruktor `typeid(matrix)` zu übergeben und die Implementierungsdetails verborgen zu halten.

Der Grund warum eine explizite Übergabe zwingend nötig ist (statt einfach optional) ist aber, dass die Entität zum Zeitpunkt an dem der Konstruktor von `entity` läuft noch nicht im Ansatz vollständig konstruiert ist und deshalb noch keine RTTI verfügbar ist. Entsprechend ist es hier noch nicht möglich den finalen Typen zu erfragen. Eine Folge dieser Anforderungen ist es, dass `entity` keinen Default-Konstruktor besitzt und entsprechend immer explizit erzeugt werden muss. Hierbei handelt es sich jedoch nicht um eine technisch erzwungene Entscheidung, sondern um etwas, was prinzipiell jederzeit geändert werden könnte, sollte es notwendig werden.

Da in der Ontologie später nur Zeiger auf `const entity` gespeichert werden, ist es für einige Operationen wie etwa Anfragen nach Entitäten die bestimmte Bedingungen erfüllen notwendig, dass `entity` eine Klasse mit mindestens einer virtuellen Funktion ist. Hierfür bietet sich der Destruktor an, der nach weithin akzeptierten Programmierstandards sowieso beim Vorhandensein auch nur einer virtuellen Methode selbst virtuell sein muss. Die Tatsache, dass ohnehin ein Destruktor definiert werden muss, der die Instanz im Falle einer gebundenen Entität aus der Ontologie entfernt, kann darüber hinaus als weiteres Argument für diese Entscheidung gelten.

Die explizite Implementierung des Move-Konstruktors und des Move-Zuweisungsoperators (hier nicht gezeigt) sehen so aus, dass sie der Ontologie mitteilen, wo sich die Entität nun befindet. Der RValue ist im Anschluss eine freie Entität. Der Copy-Konstruktor und der Copy-Zuweisungsoperator sind explizit gelöscht, da ihre Semantik an dieser Stelle kaum allgemein formuliert werden könnte: Zwar wäre es ohne große Probleme möglich, der Ontologie mitzuteilen, dass beide Objekte den gleichen Wert repräsentieren, allerdings wäre dies in vielen Fällen, insbesondere nach weiteren Operationen eine schlicht unwahre Aussage. Eine tiefe Kopie der zur ursprünglichen Entität gehörenden Axiome wäre ebenfalls möglich, hätte aber ähnliche Probleme.

Um gebundene Entitäten einer Ontologie eindeutig identifizieren zu können, besitzt jede `entity` noch ein Attribut `m_id`, welches über die Methode `id()` angefragt werden kann. Diese Nummer ist innerhalb einer jeden Ontologie zu jedem Zeitpunkt eindeutig. Sie könnte aber über die Lebenszeit der Ontologie für verschiedene, nicht zum gleichen Zeitpunkt existierende Entitäten benutzt werden, wenn etwa eine Form der automatischen Ressourcenbereinigung (Garbage Collection) implementiert würde, dies ist jedoch gegenwärtig nicht der Fall. Näheres zu dieser ID wird im Teilkapitel über die `ontology`-Klasse beschrieben werden.

Zu guter Letzt, dient die `name()`-Methode dazu, eine für den Beweiser geeignete textuelle Repräsentation der Entität zu erzeugen. Diese ist schlicht als „`o_`“ konkateniert mit der dezimalen Darstellung der ID definiert und wird entsprechend zurückgegeben.

Nun ergibt sich die Problematik, dass es nicht immer möglich ist, eine geeignete Basisklasse für seine Typen zu erzeugen, da diese etwa Teil einer existierenden Bibliothek sein könnten, oder, noch problematischer, eingebaute C++-Typen sind, von denen noch nicht einmal geerbt werden kann. Um diese Probleme zu entgehen wurde das Wrappertemplate `e<T>` geschrieben, welches im Prinzip nichts anderes tut, als eine Instanz `T` in einer von `entity` erbenden Klasse zu speichern und über verschiedenen Methoden Zugriff auf diese zu gewähren:

```
template <typename T>
class e : public entity {
public:
    template <typename... Args>
```

```
    e(ontology& o, Args&&... args)
            : entity{o, typeid(*this)},
              m_value{std::forward<Args>(args)...} {}
    template <typename... Args>
    e(std::nullptr_t, Args&&... args)
            : entity{nullptr},
              m_value{std::forward<Args>(args)...} {}
    T& operator*() { return m_value; }
    const T& operator*() const { return m_value; }
    T* operator->() { return &m_value; }
    const T* operator->() const { return &m_value; }

    private:
    T m_value;
};
```

Da hier nun der exakte Typ der Instanz bekannt ist, wird dieser automatisch an den `entity`-Konstruktor übergeben wenn eine gebundene Entität erzeugt werden soll. Darüber hinaus existiert wieder ein Konstruktor für freie Entitäten, der `nullptr` als Argument akzeptiert. Beide Konstruktoren nehmen darüber hinaus einen Parameter-Pack als Argument, den sie per perfect-forwarding an den Konstruktor von `T` weiterleiten.

`T` wird direkt als Wert in `e<T>` gespeichert, was den großen Vorteil höherer Kompaktheit und damit verringertem Laufzeitoverhead mit sich bringt, aber die Benutzung dynamischer Vererbung praktisch ausschließt. Diese Beschränkung ist jedoch von geringerem Nachteil, da es ja nach wie vor möglich ist, direkt von `entity` zu erben und das gewünschte Verhalten, wie auch immer es aussehen mag, zu erzeugen.

Die speziellen Methoden sind diesmal nicht explizit definiert, da das von `entity` gelieferte Verhalten bereits alles tut was nötig ist. Dies bedeutet auch, dass `e<T>` niemals kopierbar ist, aber move-bar, so denn `T` einen Copy/Move-Konstruktor bzw. -Zuweisungsoperator definiert.

Der Zugriff auf das `T` einer Instanz von `e<T>` erfolgt über den Dereferenzierungs- und den Pfeiloperator, wie sie von Zeigern bekannt sind. Eine implizite Typkonvertierung wäre zwar möglich, würde aber bei der Verwendung von Methoden kaum helfen und könnte je nachdem wie Funktionen die ein `T` nehmen überladen sind, insbesondere im Zusammenhang mit Templates, merkwürdige Effekte haben, weswegen dies nicht implementiert wurde. Es handelt sich hierbei jedoch um eine Designentscheidung, nicht um eine technische Limitierung des Ansatzes.

Zuletzt wurden für `e<T>` noch geeignete Vergleichsoperatoren und eine Spezialisierung von `std::hash` definiert, welche einfach auf die Implementierungen des zugrunde liegenden Typen `T` zugreifen.

4.3 Architektur der Formeln und Prädikate

Eine Formel stellt in SOOP eine prädikatenlogische Aussage dar, die sich aus Prädikaten, Entitäten (sowohl Typen als auch Laufzeitvariablen) und (in der Regel von Quantoren erzeugten) Variablen zusammensetzt.

Um etwa auszudrücken, dass der Vergleich mit einem Prädikat l transitiv ist, sofern die Argumente Instanzen eines Typs I sind, lässt sich prädikatenlogisch schreiben:

$$\forall a, b, c : a \in I \land b \in I \land c \in I \implies (l(a, b) \land l(b, c) \implies l(a, c))$$

Im Kontext von SOOP wären hier a, b und c Variablen, I eine Typ-Entität und $\land$ (in SOOP: `and_`), $\in$ (`instance_of`), $\implies$ (`implies`) und l reguläre Prädikate.

Anders als gelegentlich in der Prädikatenlogik gesehen, werden Typen in SOOP nicht als einstellige Prädikate dargestellt, da sich so auch ohne Prädikatenlogik höherer Ordnung leichter Aussagen über Typen treffen lassen.

$\forall$ (`forall`) stellt neben $\exists$ (`exists`) einen der beiden Quantoren dar. Diese sind zwar technisch als Prädikate implementiert, es sei jedoch angemerkt, dass es sich hierbei um ein Implementierungsdetail handelt.

Formeln setzen sich nun aus Teilformeln zusammen die über Prädikate verbunden werden, wobei auf der tiefsten Ebene nur Entitäten und Atome zulässig sind, während die höchste Ebene immer aus einem instanziierten Prädikat besteht (theoretisch wäre auch eine Formel denkbar, die nur aus einem boolschen Atom besteht, dies ist in SOOP jedoch mangels Entitäten boolschen Typs weder möglich noch sinnvoll).

In SOOP sind Prädikate nun in der Regel als Kombination eines Typen und einer Funktion aufgebaut, wobei es sich bei dem Typ in der Regel um eine Instanziierung eines Klassentemplates handelt welches die Argumente und den Namen des Prädikats speichert und bei der Funktion um eine Erzeuerfunktion des fraglichen Typs um die Templateargumente zu inferieren. Ein vereinfachtes Beispiel könnte etwa wie folgt aussehen:

```
namespace preds {

template<typename L, typename R>
struct my_and_t : soop::is_predicate{
    my_and_t(const L& l, const R& r): lhs{l}, rhs{r} {}

    void collect_entities(std::vector<std::size_t>& ids);
    void stream(std::ostream& out,
                const std::vector<std::string>& names) const;

    L lhs;
    R rhs;
};

template<typename L, typename R>
auto my_and(const L& l, const R& r)
-> my_and_t<to_bound_type<L>, to_bound_type<R>> {
    return {l, r};
}

} // namespace preds
```

Zunächst sind hier die empfohlenen Namenskonvention zu sehen:

Die Funktion trägt den Prädikatsnamen so wie er im Code verwendet werden soll während der Typ diesen Namen um ein `_t` ergänzt. Sowohl die Funktion als auch das Template befinden sich im Namensraum `preds`, welcher ein Unternamensraum des Hauptnamensraums der Quellbibliothek sein sollte. Die aktuelle Implementierung verzichtet darauf, `preds` zu einem `inline`-Namensraum zu machen, dies ist jedoch nicht zwingend für jede andere Bibliothek sinnvoll.

Aus technischer Sicht ist das Funktionstemplate `my_and` als analog zu anderen der Typinferenz dienenden Funktionen wie `std::make_pair` und `std::make_tuple` zu verstehen, auch wenn über das später besprochene `to_bound_type` noch eine gewisse Aufbereitung der Argumente stattfindet.

Der interessante Teil findet nun im Klassentemplate `my_and_t` statt: Hier werden nun die Argumente direkt als Werte gespeichert, was die großen Vorteile erhöhter Kompaktheit der Prädikate, Verzicht auf unnötige Speicheralokationen und komplette Verfügbarkeit der Typinformationen hat.

Um für SOOP verwendbar zu sein, müssen Prädikate zwei Methoden implementieren: `stream` und `collect_entities` mit den im Beispiel gezeigten Signaturen. Die Methode `stream` ist letztlich nichts anderes als eine print-Funktion, deren zweites Argument für Prädikate nicht weiter Interessant ist und entsprechend nur durchgereicht wird (eine nähere Beschreibung erfolgt später).

Der Effekt von `stream` muss nun daraus bestehen, den Namen des Prädikats und seine Argumente als LISP-artige S-Expression auf den übergebenen Ausgabestrom zu schreiben. Eine Implementierung für `my_and_t` könnte daher wie folgt aussehen:

```
template<typename L, typename R>
void my_and_t<L, R>::stream(
    std::ostream& out,
    const std::vector<std::string>& names
) const {
    out << "(my_and ";
    lhs.stream(out, names);
    out << " ";
    rhs.stream(out, names);
    out << ")";
}
```

Angenommen, `lhs` wird als „l“ ausgegeben und `rhs` als „r“, so würde die Ausgabe also „`(my_and l r)`“ lauten.

Dadurch, dass die `stream`-Methode wiederum die `stream`-Methoden der Prädikatsargumente aufrufen, wird eine LISP-artige, textuelle Repräsentation des Ausdrucksbaums errichtet, welche als Eingabe für den Beweiser geeignet ist.

Was `collect_entities` angeht, ist es wie beim `names`-Argument von `stream` so, dass diese Funktion nicht für Prädikate existiert und entsprechend hier nur dahingehend besprochen wird, dass sie so zu implementieren ist, dass in ihr das globale Funktionstemplate `collect_entity` mit jedem Argument einmal in der selben Reihenfolge wie in `stream` aufzurufen ist:

```
template<typename L, typename R>
void my_and_t<L, R>::collect_entities(
    std::vector<std::stream>& ids
) const {
    collect_entity(lhs, ids);
    collect_entity(rhs, ids);
}
```

Im Gegensatz zu den komplex typisierten Prädikaten ergeben sich bei den über reguläre Vererbung bereitgestellten Entitäten andere Probleme.

Da sämtliche Entitäten von `soop::entity` erben müssen, ist hier prinzipiell weniger Aufwand für die Implementierung nötig. Im Grunde wird ein allgemeiner Typ `bound_entity` definiert welcher eine die Entität identifizierende Ganzzahl speichert und die `stream`- und `collect_entity`-Funktionalitäten bereit stellt:

```
struct bound_entity {
    bound_entity(const entity& e):
        id{e.id()} {}
    void stream(std::ostream& out,
                const std::vector<std::string>& names) const;
    std::size_t id; // explanation below
};

void collect_entity(std::vector<std::size_t>& ids,
                    bound_entity& v);
```

Entitäten sind der Grund für die `names` und `ids` Argumente der vorher bezeichneten Funktionen. Aus Gründen die bei der Beschreibung der `formula`-Klasse näher erläutert werden (im Grunde geht es darum Änderungen zu erleichtern), wird die Identität der benutzten Entitäten weder per Zeiger, noch per ID direkt im Expression-Template gespeichert, sondern in einem `std::vector<size_t>`, der sich außerhalb des Baums befindet und die IDs enthält. Im Baum selbst wird wiederum der Index der ID in besagtem `std::vector` hinterlegt. Da sowohl die Entitäts-ID, als auch der Index als `std::size_t` repräsentiert werden und niemals beide zugleich einen gültigen Wert enthalten, existiert in der `bound_entity`-Klasse nur ein Feld für beide Werte, welches zunächst mit der ID der referenzierten Entität initialisiert wird, bevor diese von `collect_entity` durch den Index ersetzt wird. Entsprechend ist `collect_entity` wie folgt definiert:

```
void collect_entity(std::vector<std::size_t>& ids, bound_entity& v) {
    const auto index = ids.size();
    ids.push_back(v.id);
    v.id = index;
}
```

Da nun endgültig kein direkter Zugriff mehr auf die Entität möglich ist, wird für die Ausgabe ein `std::vector<std::string>` mit für die Ausgabe geeigneten texutellen Repräsentationen der Entitäten übergeben. Entsprechend schreibt `stream` für Entitäten lediglich diese Zeichenkette auf den Ausgabestrom:

```
void bound_entity::stream(std::ostream& out,
                          const std::vector<std::string>& names
) const {
    out << names.at(id);
}
```

Obwohl Typen im Kontext von SOOP auch Entitäten darstellen, werden diese gegenwärtig getrennt von diesen behandelt, da die Verwendung sich in der Praxis signifikant unterscheiden dürfte.

Das Analogon zu `bound_entity` ist für Typen die Klasse `bound_type`, welche wie folgt definiert ist:

```
class bound_type {
public:
    bound_type(const std::type_info& info) : m_type{info} {}
    void stream(std::ostream& out,
                const std::vector<std::string>&) const;

private:
    std::type_index m_type;
};

inline void collect_entity(std::vector<std::size_t>&,
                           const bound_type&) {}
```

Da Typen als Entitäten Atome darstellen, aber im Gegensatz zu regulären Entitäten nicht zentral in der `formula`-Klasse referenziert werden, kann `collect_entity` als leere Funktion definiert werden.

`bound_type` selbst enthält eine Instanz von `std::type_index`, welche Typinformationen zum gespeicherten Typen enthält und im Konstruktor von einer `std::type_info` initialisiert wird, welche in der Regel mittels des `typeid`-Operators gewonnen wurde.

Die textuelle Repräsentation, die von der `stream`-Methode erzeugt wird, entspricht schließlich einfach der Ausgabe von `type_index::name()`, bei welcher es sich auf den betrachteten Plattformen um den Namen des Typs nach Anwendung von „Name-Mangling“[1] handelt, wodurch dieser auch gleich den Vorteil hat, eindeutig zu sein.

Um die Verwendung von Typen in Formeln einfacher zu gestalten existiert zuletzt noch ein Variablentemplate `soop::type`:

```
template <typename T>
static auto type = bound_type{typeid(T)};
```

Hiermit können dann Formeln wie die folgende definiert werden:

```
// assume the existance of a predicate subclass_of
subclass_of(type<base>, type<derived>);
```

[1] „Name-Mangling“ beschreibt in C++ das Erzeugen eines eindeutigen Namen aus der Signatur einer bestimmten Funktion oder eines bestimmten Typs.

Die letzten von SOOP benutzten Primitive sind Variablen. Hierbei handelt es sich einfach um Instanziierungen des Klassentemplates `variable`, welches analog zu folgendem Code definiert ist:

```
template <char... Name>
struct variable {
    static std::string str() { return {Name...}; }
    static void stream(std::ostream& out,
                       const std::vector<std::string>&) {
        out << str();
    }
};

template <char... Name>
void collect_entity(std::vector<std::size_t>&, variable<Name...>) {}
```

Wie auch bei Typentitäten handelt es sich hier um Atome die nicht von `formula` verwaltet werden, weswegen auch hier `collect_entity` als leere Funktion definiert werden kann.

Auch die Ausgabe und die `str`-Methode folgen den bisherigen Mustern in einem Ausmaß das keine weitere Erklärung nötig macht.

Die Idee hinter `variable` ist ein Klassentemplate zu schaffen, welchem über seine Template-Argumente ein Name zugewiesen wird, und in Formeln dann auch schlicht als solcher verwendet wird.

Der primäre Anwendungsfall für Variablen ergibt sich in Kombination der als Prädikate implementierten Quantoren: Um etwa die folgende Aussage (ungeachtet ihres Wahrheitswertes)

$$\forall x, y : x = y$$

nach SOOP zu übersetzen, ist es nötig zwei lokal gebundene Variablen zu erzeugen und diese an `forall` zu übergeben. Um hier eine angenehme Syntax zu erzeugen wurde das Helfertemplate `bound_vars` geschrieben:

```
class bound_vars {
public:
    template <typename... Args>
    bound_vars(Args... args);
    const std::string& str() const { return m_str; }

private:
    std::string m_str;
};

template <typename... Args>
bound_vars::bound_vars(Args... args) {
    static_assert(sizeof...(Args) > 0, "");
```

```
    m_str = "(";
    tuple_foreach(std::tie(args...), [&](const auto& arg) {
        m_str += "(";
        m_str += arg.str();
        m_str += " Entity) ";
    });
    m_str += ')';
}
```

Aktuell nicht statisch abgefangen, aber per Dokumentation verlangt ist dass sämtliche Argumente Instanzen von `variable` sind.

Die Grundidee hinter `bound_vars` ist es, bereits im Konstruktor alle Variablen in eine für den Beweiser geeignete, textuelle Form zu bringen und diese dann bei Bedarf einfach auszugeben. Das Format sieht hierbei so aus, dass jeder Bezeichner gemeinsam mit dem Typnamen „Entity“ in Klammern geschrieben wird und diese Liste dann wiederum in ein weiteres Paar Klammer. Sind die Variablennamen also „`x`“ und „`y`“, so lautet die Ausgabe „`((x Entity) (y Entity) )`“.

Das in der Implementierung benutzte Helferfunktionstemplate `tuple_foreach` stellt im Prinzip eine herkömliche Schleife dar, welche eine Funktion mit jedem Element eines Tupels als Argument einmal in sequentieller Reihenfolge aufruft.

Da der Konstruktor von `bound_vars` implizit ist, kann nun ein Quantor dadurch geschrieben werden, dass er als erstes Argument eine Instanz von `bound_vars` entgegennimmt, welche über geschweifte Klammern erzeugt wird und die eigentliche Aussage im Anschluss. Obige Formel kann also wie folgt ausgedrückt werden:

```
const soop::variable<'x'> x; // recommended way of use
const soop::variable<'y'> y; // for more convenience
soop::preds::forall({x, y}, soop::preds::equals(x, y));
```

Mit der Kombination all dieser Werkzeuge lassen sich nun zwar fast beliebig komplexe Formeln ausdrücken, allerdings ist es aufgrund der stark verschiedenen Typen weder möglich diese direkt in einem `std::vector` zu speichern, noch sind Fehlermeldungen mit auch nur einer Formel ansehnlich: Mehrere Formeln in einer Fehlermeldung erscheinen daher für Benutzer der Bibliothek nicht zumutbar. Aus diesen Gründen wurde die vereinigende Klasse `function` verfasst, welche eine komplexe Formel via Typauslöschung[2] so speichert, dass die entscheidende Funktionalität verfügbar bleibt, gleichzeitig die Probleme aber umgangen werden:

```
class formula {
public:
    formula() = default;

    template <typename P>
    formula(P p);
```

[2] Typauslöschung beschreibt eine Technik bei der ein Klassentemplate von einer Basisklasse so erbt, dass die interessante Funktionalität über virtuelle Methoden verfügbar bleibt, während die Instanz selbst nur noch über die polymorphe Basisklasse angesprochen werden kann.

```
    std::string to_string() const;

    explicit operator bool() const { return m_formula != nullptr; }

private:
    // this is just standard type-erasure:
    class basic_formula {
    public:
        virtual ~basic_formula() {}
        virtual void stream(
            std::ostream&,
            const std::vector<std::string>&) const = 0;
    };
    template <typename P>
    class concrete_formula : public basic_formula {
    public:
        concrete_formula(P p) : m_pred{std::move(p)} {}

        void stream(
            std::ostream& s,
            const std::vector<std::string>& args)
                const final override;

    private:
        P m_pred;
    };

    std::unique_ptr<basic_formula> m_formula;
    std::vector<std::size_t> m_args;
};
```

Konkret funktioniert die Typauslöschung hier wie folgt: Die abstrakte Basisklasse `formula::basic_formula` definiert einen virtuellen Destruktor und eine rein virtuelle `stream`-Methode mit dem bereits bekannten Interface. Hiervon erbt das Klassentemplate `formula::concrete_formula`, welches die eigentliche Formel mit ihrem kompletten Typen als normalen Wert enthält. Durch die hier wieder verfügbare Typisierung kann nun auch die `stream`-Methode einfach dadurch implementiert werden, dass sie ihre Argumente an die `stream`-Methode der eigentlichen Formel weiterleitet.

`formula` selbst speichert nun einfach einen `std::unique_ptr<basic_formula>`, welcher auf eine im Konstruktor erzeugte Instanz einer `concrete_formula` zeigt. Dies wird dadurch ermöglicht, dass der Konstruktor ein Methodentemplate ist und daher ebenfalls den exakten Typen der eigentlichen Formel kennt. Ist dieser Typ `P`, so kann nun einfach ein geeigneter `std::unique_ptr` erzeugt werden der dann `m_formula` zugewiesen wird:

```
m_formula = std::make_unique<concrete_formula<P>>(std::move(p));
```

Allerdings ist es vorher noch nötig alle Laufzeitentitäten einzusammeln; dies findet ebenfalls noch im Konstruktor von `formula` statt, so dass es nicht nötig ist, das Interface von `basic_formula` weiter aufzublähen.

Die gesamte Implementierung des Konstruktors sieht daher so aus:

```
template <typename P>
formula::formula(P p) {
    p.collect_entities(m_args);
    m_formula = std::make_unique<concrete_formula<P>>(std::move(p));
}
```

Im Anschluss enthält `m_args` nun die IDs sämtlicher in der Formel vorkommender Laufzeitentitäten in einer für jegliche Weiterverarbeigung geeigneten Form. Sollte etwa eine Form der automatischen Ressourcenverwaltung eingeführt werden, die IDs neu zuweist, so wäre dies sehr leicht dadurch möglich, dass sämtliche IDs in `m_args` naiv umbenannt werden.

Die Klasse `formula` selbst definiert keine speziellen Memberfunktionen und kann daher wie der `std::unique_ptr` den sie enthält nicht kopiert, aber verschoben werden. Letzteres ist auch dadurch effizient, dass diese Operation auch für den enthaltenen `std::vector` sehr billig ist.

4.4 Architektur der Ontologie

Die `ontology`-Klasse ist das Herzstück von SOOP. Sie verwaltet sämtliches Wissen über Typen, Prädikate, Entitäten und Axiome.

Für ihr Verständnis ist die Kenntnis folgender Helfer erforderlich:

```
template<typename... Types>
struct type_list_t{};
template<typename... Types>
type_list_t<Types...> type_list;

template<template<typename...>class... Preds>
class pred_list{};

using axiom_list = std::vector<formula>;

struct not_found_error: std::runtime_error {
    using std::runtime_error::runtime_error;
};
```

`type_list_t` ist hierbei ein Klassentemplate, dass eine statisch bekannte Liste von Typen enthält, welche über das Variablentemplate `type_list` bequem erzeugt werden kann. `pred_list` ist das Analogon für Prädikate und `axiom_list` eine dynamische Liste von Formeln.

`not_found_error` ist eine Exception deren Bedeutung an der Stelle ihrer Anwendung beschrieben wird.

Mit diesem Wissen ist nun eine Betrachtung der Grundstruktur von `ontology` möglich:

```
class ontology {
public:
```

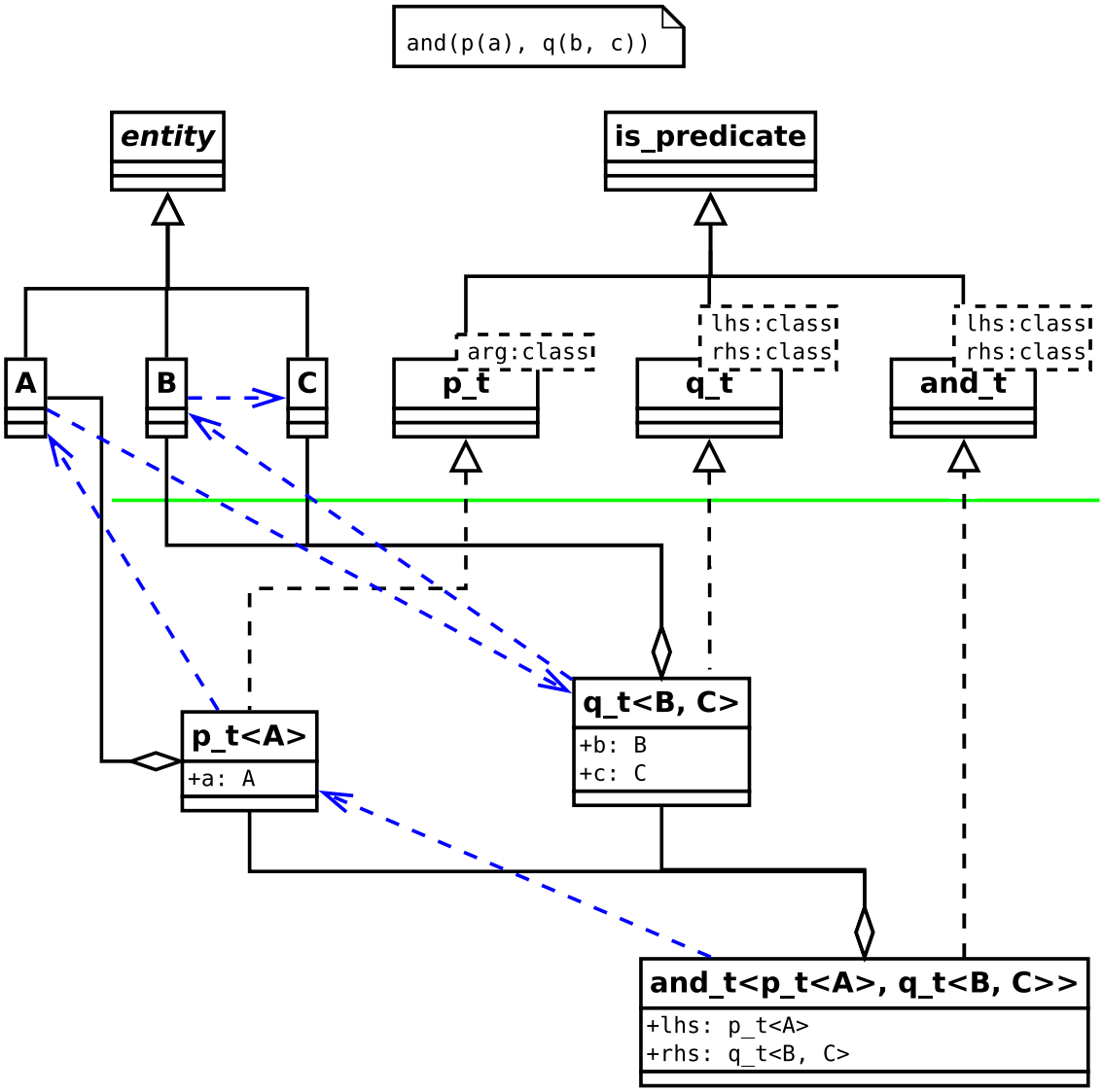

Abbildung 4.2: Instanziierung eines Prädikats.
Oberhalb der grünen Linie ist die explizit implementierte Klassenstruktur der Atome zu sehen. Unterhalb wird demonstriert wie diese zu einem komplexen Objekt zusammengebaut werden, ohne dass hierfür virtuelle Funktionen oder Vererbung nötig wären.
Die Traversierungsreihenfolge des Baumes für `stream` und `collect_entities` wird hier durch die blauen Pfeile signalisiert; es handelt sich im Kern um eine einfache Tiefensuchen mit deterministischer Reihenfolge.

```
    ontology();
    template <typename... Ts, template<typename...>class... Ps>
    ontology(type_list_t<Ts...>, pred_list<Ps...> ps,
             axiom_list as = {});

    std::size_t add_axiom(formula axiom);

    std::size_t add_entity(entity& e);
    std::size_t add_entity(entity& e, const std::type_info&);
    void delete_entity(std::size_t id);

    bool check_sat() const;
    bool request(const formula& conjecture) const;

    template<typename...Ts, typename... Vars>
    std::tuple<const Ts&...> request_entities(
        const formula& description, Vars...) const;

    template<template<typename...>class P>
    void ontology::add_predicate();

    // several public methods ommited
private:
    // several private methods ommited

    axiom_list m_axioms;
    std::vector<std::pair<
        const entity*, std::vector<std::size_t>>> m_entities;
    std::unordered_set<std::string> m_known_types;
    std::vector<std::pair<std::string, std::size_t>> m_predicates;
};
```

Die Liste aller bekannten Axiome, `m_axioms`, kann, da Axiome leer sein können, leere Einträge enthalten (etwa weil das entsprechende Axiom gelöscht wurde), ist aber ansonsten nicht weiter interessant.

Die Liste aller Laufzeitentitäten `m_entities` stellt sich im Vergleich als erheblich komplexer dar: Bei ihr handelt es sich um einen `std::vector` der konstante Zeiger auf die jeweilige Entität und einen `std::vector` der Axiom-Indices enthält, an denen die jeweilige Entität beteiligt ist.

Die ID der Entitäten stellt nichts anderes als den Index in dieser Liste dar. Wird nun eine Entität aus der Ontologie gelöscht, so werden auch alle Formeln in denen sie enthalten ist entfernt.

Im Gegensatz zu diesem vergleichsweise komplexen Modell, werden bei Typentitäten nur die Namen als `std::string` in einem Hashset gespeichert. Grund für den Unterschied ist, dass Änderungen der beteiligten Typen zur Laufzeit kaum einen realistischen Anwendungsfall finden, da sie nicht zur Laufzeit erzeugt werden können und auch sonst praktisch konstant sind, was im deutlichen Gegensatz zum Verhalten von Laufzeitentitäten steht. Entsprechend

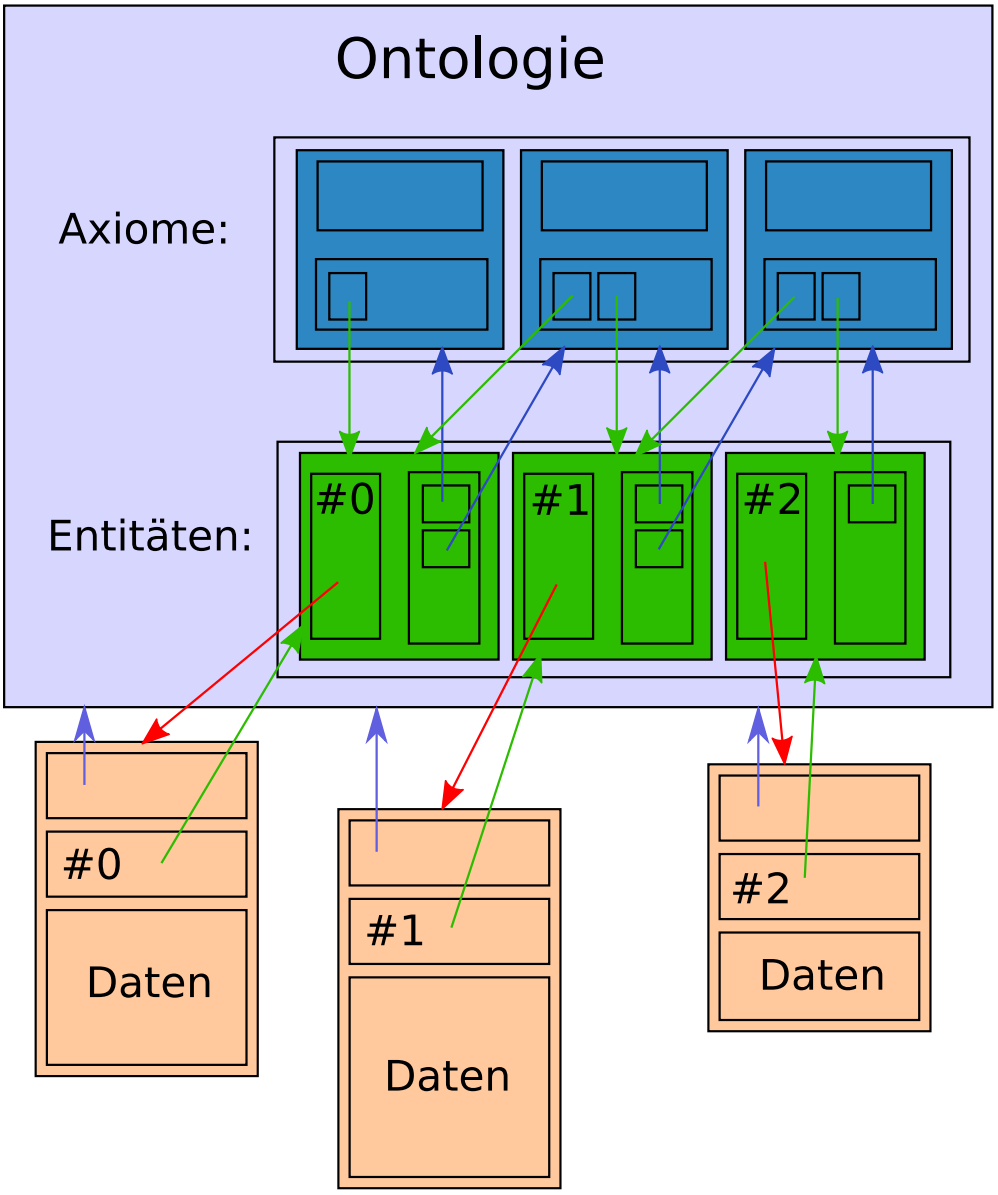

Abbildung 4.3: Verbindung von Entitäten mit der Ontologie.
Alle Entitäten enthalten einen Zeiger auf die Ontologie (blaue Pfeile) und den Index des sie beschreibenden Objekts im Entitäten-Container (untere grüne Pfeile). Diese Objekte wiederum enthalten einen Zeiger auf die Entitäten (rote Pfeile) und eine Liste der Indizes der sie betreffenden Axiome im Axiom-Container (cyanfarbene Pfeile). Die Axiome wiederum speichern eine Liste der Indizes der von ihnen betroffenen Enitäten (obere grüne Pfeile).

ist es lediglich nötig, dass die Ontologie die Namen kennt um sie dem Beweiser mitzuteilen, wofür ein Hashset als ideale Wahl erscheint.

Zuletzt werden die Namen aller Prädikate mit ihrer Arität in `m_predicates` gespeichert. Der Grund warum hier ebenfalls keine semantischeren Daten vorliegen ist analog zu dem bei den Typentitäten genannten Grund: Der praktische Nutzen wäre äußerst fraglich.

Ist nun eine Ontologie mit den gewünschten Daten erstellt, so kann ihre Konsistenz über die Methode `check_sat` erfragt werden, welche die gesamten verfügbaren Daten in eine für den Beweiser geeignete Textform bringt und diese mit dem Befehl „`(check-sat)`“ konkateniert, welcher Z3 dazu auffordert die Erfüllbarkeit aller Daten zu überprüfen. Diese Zeichenkette wird nun an die `try_proof`-Funktion des Beweisermoduls übergeben, dessen boolsche Antwort der Rückgabewert der Funktion ist. Die Methode `request` tut im Prinzip exakt das Gleiche, nur dass sie das Übergeben einer zusätzlichen Formel erlaubt, welche nicht Teil der Ontologie wird, sondern nur im Rahmen der einen Anfrage Verwendung findet.

Beide Methoden ermöglichen bereits vergleichsweise weitgehende Anfragen etwa zur Korrektheit des Programmzustandes, allerdings sind sie nicht dazu in der Lage, Lösungen zu erfragen, was die interessanteste Anwendung von Ontologien darstellt. Hierfür steht die Methode `request_entities` bereit, welche eine Formel mit ungebundenen Variablen, die Liste dieser Variablen und ihre erwarteten Typen entgegennimmt und versucht die Variablen so zu belegen, dass die jeweilige Formel erfüllt wird. Im Anwendungsbeispiel:

```
// o is the ontology that already
// contains all the needed state:
assert(o.check_sat());
// ‘min‘, ‘max‘ and ‘x‘ are ‘soop::variable‘s and
// we have imported all needed namespaces:
const auto answer = o.request_entities<e<int>, e<int>>(and_(
    forall({x}, or_(less(min, x), equal(min, x))),
    forall({x}, or_(less(x, max), equal(max, x)))),
    min, max);
const auto& smallest = std::get<0>(answer);
std::cout
    << *smallest
    << "is the smallest element by some measure ‘less‘.\n";
```

Der Rückgabewert von `request_entities` ist ein Tupel von konstanten Referenzen auf die für die Lösung verwendeten Entitäten. Die benutzten Typen werden als Templateargumente übergeben und münden in einer Exception falls sich herausstellt, dass sie nicht mit den Ergebnistypen übereinstimmen.

Die Erzeugung dieses Ergebnisses findet in mehreren Schritten statt:

- Als Erstes werden die (zuletzt übergebenen) ungebundenen Variablen zu Zeichenketten konvertiert die in einem `std::vector` gespeichert werden, was die weitere Verarbeitung erleichtert.
- Der nächste Schritt verläuft ähnlich den analogen Schritten von `check_sat` und `request` dahingehend dass nun die gesamte Ontologie in eine Textform gebracht wird; im Detail existiert jedoch der Unterschied, dass darüber hinaus Entitäten mit den Namen der Variablen deklariert werden, diese jedoch explizit nicht als von anderen

Variablen verschieden deklariert werden und die übergebene Formel ähnlich wie in der `request`-Implementierung auch zum Text, nicht aber zur Ontologie hinzugefügt wird.

- Diese textuelle Problembeschreibung wird nun gemeinsam mit der Liste der Lösungsvariablennamen an die freie Funktion `soop::request_entities` übergeben, welche einen `std::vector` der für die Belegung verwendeten Entitäts-IDs zurück gibt. (Hat eine Lösungsentität keine ID wird stattdessen `SIZE_MAX` verwendet, was meist auf einen Fehler hindeutet.)
- Dieser ID-Vector wird nun zu einem Vector von konstanten Entitätszeigern konvertiert, wobei im Falle nicht existierender IDs Nullzeiger verwendet werden.
- Wenn der resultierende `std::vector<const entity*>` nun Nullzeiger beliebiger Herkunft (also auch von gelöschten Entitäten) enthält, wird eine Instanz von `not_found_error` geworfen.
- Zuletzt werden nun alle Zeiger dereferenziert und dann mittels eines `dynamic_cast` zu den gewünschten typisierten konstanten Referenzen konvertiert, welche in einem `std::tuple` zurückgegeben werden.

Es ist in diesem Kontext wichtig zu verstehen, dass die Lebensdauer der von den zurückgegebenen Referenzen referenzierten Variablen nicht von SOOP kontrolliert wird und es Aufgabe des Programmierers ist, sicherzustellen, dass das Ergebnis entsprechend nicht mehr benutzt wird, wenn die fraglichen Entitäten nicht mehr existieren.

Zur Verwaltung von Laufzeitentitäten existieren zwei grundlegende Operationen: `add_entity` und `delete_entity`, welche Entitäten zur Ontologie hinzufügen, respektive löschen.

Das Hinzufügen funktioniert hierbei so, dass eine neue Entität in `m_entities` eingefügt wird und ein neues Axiom in `m_axioms` welches besagt, dass die Entität eine Instanz des in der `std::type_info` übergebenen Typen ist. Wird keine Typinformation übergeben, so wird stattdessen der `typeid`-Operator benutzt, welcher jedoch nicht in Konstruktoren funktioniert. Beim Hinzufügen des Axioms wird außerdem darauf geachtet, die Axioms-ID in der Liste der von der Entität abhängigen Axiome zu speichern. Die Hauptschwierigkeit bei der Implementierung dieser Funktionalität stellt dass Sicherstellen transaktionellen Verhaltens dar, wofür zuerst alle Operationen ausgeführt werden die Exceptions werfen können, aber den semantischen Zustand der Ontologie nicht beeinflussen, bevor tatsächliche Änderungen durchgeführt werden. Der Rückgabewert von `add_entity` ist schließlich die ID die der Entität zugewiesen wurde. Diese kann im Folgenden nun auch dazu benutzt werden um die Entität mittels `delete_entity` wieder zu entfernen. Dies funktioniert nun so, dass alle Axiome die die Entität direkt betreffen zurückgesetzt („gelöscht“) werden und im Anschluss der Eintrag in `m_entities` durch einen Default-konstruierten ersetzt wird, der an keinen Axiomen beteiligt ist und auf keine Entität zeigt.

Die Verwaltung von Axiomen funktioniert ähnlich: Mittels `add_axiom` kann eine Formel als Axiom hinzugefügt werden. Die Vorgehensweise sieht hier so aus, dass die Formel zu `m_axioms` hinzugefügt wird und bei allen der an der Formel beteiligten Entitäten der Verweis auf das neue Axiom hinzugefügt wird. Auch hier wird transaktionelles Verhalten garantiert. `add_axiom` gibt einen `std::size_t` zurück, der diesmal die ID der gerade hinzugefügten Formel repräsentiert. Diese IDs sind unabhängig von den IDs der Entitäten und dürfen nicht vertauscht werden. Das Löschen von Axiomen sieht nun so aus, dass das Axiom mit der jeweiligen ID von `delete_axiom` einfach nur zurückgesetzt wird, ohne das es aus den Listen

der verwendeten Entitäten gelöscht wird. Der Hintergrund hiervon ist der vergleichsweise hohe Aufwand in allen Listen nach der fraglichen ID zu suchen, während der Nutzen einer solchen Löschung zumindest ohne automatische Ressourcenbereinigung kaum zu erkennen wäre.

Das Hinzufügen von Prädikaten funktioniert über das Methodentemplate `add_predicate` welches ein Klassentemplate als einziges Templateargument erhält und dieses mit `soop::get_meta_information` instanziiert. Diese Instanziierung darf nicht fehlschlagen und muss zwei statische Methoden `name` und `rank` bereitstellen, die den Namen und den Rang des Prädikats zurückgeben. Diese Werte werden nun zu `m_predicates` hinzugefügt:

```
template<template<typename...>class P>
void ontology::add_predicate() {
    using Pred = P<get_meta_information>;
    m_predicates.emplace_back(Pred::name(), Pred::rank());
}
```

Im Quelltext nicht gezeigt sind darüber hinaus in der eigentlichen Implementierung noch einige weitere Methoden für Dinge wie das Hinzufügen von Typentitäten, deren detaillierte Erleuterung jedoch keinen großen Erkentnisgewinn brächte.

Die händisch implementierten Move-Operationen und der Destruktor hingegen verfügen über Besonderheiten, die über Standardverhalten hinaus gehen: Konstruktion und Zuweisung stellen beide die Verfügbarkeit von `instance_of` sicher (siehe unten) und ändern die Ontologie-Zeiger aller von der Ontologie betrachteten Entitäten so, dass sie auf den neuen Ort zeigen. Der Destruktor widerum setzt alle diese Zeiger auf Null und „befreit“ dadurch die jeweiligen Entitäten.

Zuletzt sei auf den Konstruktor eingegangen, der drei Argumentlisten erhält: Eine Liste der benutzten Typen, eine der benutzten Prädikate und eine mit Axiomen.

```
template <typename... Ts, template<typename...>class ... Ps>
ontology::ontology(type_list_t<Ts...>, pred_list<Ps...>, axiom_list as):
    m_axioms(std::move(as))
{
    using ignore = std::initializer_list<int>;
    (void) ignore{ (add_type<Ts>(),0)... };
    add_predicate<preds::instance_of_t>();
    (void) ignore{ (add_predicate<Ps>(),0)... };
}
```

Die Funktionsweise sieht nun so aus, dass zunächst die Axiome einfach zugewiesen werden und im Anschluss die Typen und Prädikate, welche über Destrukturierung der Template-Argumente erhalten werden mit den jeweiligen Methoden hinzugefügt werden. Die Hauptbesonderheit hier ist das explizite Hinzufügen von `instance_of`, welches ein Prädikat ist, dass zwar implizit in der Ontologie benutzt wird, aber nicht Teil von Z3 ist. Um unerwartete Probleme zu vermeiden wird es daher auch implizit hinzugefügt.

Bei `(void) ignore{(expression, 0)...};` handelt es sich um eine verbreitete Methode eine foreach-Schleife für Parameter-Packs zu emulieren. Die Grundidee ist hierbei den Komma-Operator zu benutzen um den Rückgabewert des Ausdrucks zu ignorieren um so

einen Pack von Nullen zu erzeugen, der von der `std::initializer_list` geschluckt wird. Der Cast nach `void` ist streng genommen nicht nötig, dient aber dazu Compilerwarnungen zu unterdrücken.

4.5 Benutzung des Beweisers

Die Schnittstelle für die Kommunikation mit dem Beweiser besteht (neben zweier Exceptions) im Kern aus den zwei, bereits im vorherigen Unterkapitel angesprochenen, Funktionen `try_proof` und `request_entities` mit den folgenden Signaturen:

```
bool try_proof(const std::string& request);

std::vector<std::size_t> request_entities(
    const std::string& problem,
    const std::vector<std::string>& results);
```

Beide Funktionen erwarten eine textuelle Beschreibung der Fragestellung in einer für Z3 geeigneten Form. Darüber hinaus nimmt `request_entities` eine Liste der Namen der zu belegenden Variablen.

Der Rückgabewert von `try_proof` is schlicht die prinzipielle Erfüllbarkeit des Beispielproblems, während es im Falle von `request_entities` die IDs der Entitäten mit denen die im `results`-Argument genannten Variablen zu befüllen sind um eine gültige Lösung zu erzeugen. Die Implementierung dieser Funktionalität sieht nun so aus, dass mittels einer Variante des Meyerschen Singletons pro Thread eine Instanz von Z3 als eigenständigem Prozess gestarted wird:

```
procxx::process& get_z3() {
    thread_local static procxx::process z3{"z3", "-in", "-nw"};
    thread_local static int dummy = [&]{
        z3.exec();
        return 0;
    }();
    (void) dummy;
    return z3;
}
```

Da Prozesse bei der Verwendung von procxx nicht automatisch gestartet werden, ist es nötig dies *einmal* nach der Konstruktion zu machen, was sich leicht mit der hier gezeigten `dummy`-Variable machen lässt, deren Initialisierung `process::exec` ausführt. Durch die Konzeption als Meyerscher Singleton ist sichergestellt, dass jeder Thread einerseits Zugriff auf einen eigenen Z3-Prozess hat, andererseits aber nicht mehr derartige Prozesse gestartet werden als nötig.

Um nun bei jeder Anfrage eine saubere Z3-Instanz zu haben, ist es nötig vor jeder Anfrage den aktuellen Beweiserzustand zu speichern und ihn am Ende wieder herzustellen, was sich mit den Befehlen `(push)` und `(pop)` erreichen lässt. Offensichtlich ist diese Situation wie geschaffen für RAII, weswegen ein entsprechender Wrapper geschrieben wurde:

```
struct z3_transaction {
    z3_transaction() {
        get_z3() << "(push)\n";
    }
    ~z3_transaction() {
        get_z3() << "(pop)\n";
        get_z3().output().sync();
    }
    // disables copy/move-ctor/assignment as well:
    z3_transaction(z3_transaction&&) = delete;
};
```

Sowohl `get_z3` als auch `z3_transaction` befinden sich in einem anonymen Namensraum der die Verwendung dieser Primitive anderswo ausschließt und somit eine robustere Implementierung unnötig erscheinen lässt.

Hiermit kann nun `try_proof` wie folgt implementiert werden:

```
bool try_proof(const std::string& problem) {
    z3_transaction transaction;
    auto& z3 = get_z3();

    z3 << problem << "(check-sat)\n";

    z3.output().sync();
    std::string line;
    std::getline(z3.output(), line);
    if (line == "sat") {
        return true;
    } else if (line == "unsat") {
        return false;
    } else {
        std::cerr << problem;
        throw no_answer_found_error{"Didn't find an answer: " + line};
    }
}
```

Es wird vorausgesetzt, dass das Problem selbst soweit fehlerfrei ist, dass von Z3 keine Fehler ausgegeben werden und auch sonst keine Ausgabe erzeugt wird. Sind diese Voraussetzungen erfüllt wird Z3 (so ein Beweis gefunden wird) „sat“ oder „unsat“ ausgeben was zum Beenden der Funktion mit dem jeweils angemessenen Rückgabewert führt. Waren die Vorausetzungen nicht erfüllt, wird die komplette Problemstellung zur Erleichterung der Fehleranalyse auf die Standardfehlerausgabe geschrieben und eine Exception geworfen.

`request_entities` ist ähnlich implementiert, beendet sich jedoch bei gegebener Erfüllbarkeit nicht (und gibt in allen anderen Fällen einen leeren `std::vector` zurück), sondern erzeugt einen Ergebnisvector der die Antwort-IDs enthält und befüllt diesen der Reihe nach dadurch, dass für jede zu befüllende Variable im `results`-Argument eine Anfrage nach dem Rückgabewert der Funktion `to-entity-id` gestellt wird wenn sie mit der jeweiligen Variablen als einziges Argument aufgerufen wird. Diese Funktion wird zuvor automatisch

von der anfragenden Ontologie so definiert, dass sie für alle Laufzeitentitäten deren ID zurückgibt und für alle anderen Argumente `SIZE_MAX`.

Diese Werte werden nun Zeilenweise von Z3 ausgegeben, von SOOP geparst und schließlich in einem `std::vector` zurückgegeben.

4.6 Helfer für Prädikatendefinition

Da die Definition eigener Prädikate mit der bisher beschriebenen Architektur als Resultat der starken Typisierung relativ viel Codeduplikation nötig macht, bietet SOOP auch Helfer um dies zu vereinfachen. Diese existieren in zwei Stufen: Das Klassentemplate `basic_predicate`, welches CRTP verwendet um die `stream`-Methode und `collect_entities`, sowie die Speicherung der Argumente bereitzustellen, als auch die Makro-Familie um `SOOP_MAKE_PREDICATE`, welche die Definition eigener Prädikate auf eine einzige Zeile Code reduzieren kann und unter anderem `basic_predicate` für die Implementierung verwendet.

Zunächst sei daher `basic_predicate` erläutert. Die Definition dieses Templates sieht wie folgt aus:

```
template <template <typename...> class Self, typename... Args>
struct basic_predicate : is_predicate {
    basic_predicate(Args... args) : args{std::move(args)...} {}
    using self = Self<Args...>;
    void collect_entities(std::vector<std::size_t>& ids);
    void stream(std::ostream& out,
                const std::vector<std::string>& names) const;
    std::tuple<Args...> args;
};
```

Die Argumente des Prädikats werden hier in einem `std::tuple` gespeichert, welches vom Konstruktor befüllt wird. Die Implementierung der `collect_entities`- und `stream`-Methoden entspricht in etwa der händischen Variante und wird durch die Verwendung des bereits erwähnten Funktionstemplates `tuple_foreach` nahezu trivial; den eigentlichen Namen des Prädikats beschafft sich das Helfertemplate aus einer vom Argument bereitzustellenden statischen Methode `name`, deren Nichtvorhandensein zu einem Fehler führt.

Um mit `basic_predicate` nun das bereits als Beispiel verwendete Prädikat `my_and` zu definieren, würde man folgenden Code schreiben:

```
namespace preds {

// we can now do it variadic as well:
template<typename... Args>
struct my_and_t : basic_predicate<my_and_t, Args...> {
    // inherit the ctor:
    using basic_predicate<my_and_t, Args...>::basic_predicate;
    static std::string() name() {return "my_and";}
};
```

```
template<typename...Args>
my_and_t<Args...> my_and(Args... args) {
    return {std::move(args)...}
}

} // namespace preds
```

Dieser Code hat zwar bereits Vorteile gegenüber der händischen Implementierung, bringt allerdings auch Nachteile mit sich: Das Erben des Konstruktors der relativ komplizierten Basisklasse ist nicht trivial, mögliche Typüberprüfungen, (hier nicht gezeigt) würden die Abfrage von Rang und Namen beim Hinzufügen des Prädikats zur Ontologie deutlich erschweren und der Gewinn in Bezug auf Codezeilen ist auch verhältnismäßig klein.

Um die benötigte Menge an Code auf ein akzeptables Niveau zu reduzieren, sind daher Makros nötig, mit denen die Erzeugung von Prädikaten wie folgt möglich ist:

```
// basic version without typechecking in C++:
SOOP_MAKE_PREDICATE(subclass_of, 2)

// advanced version, that checks the types in C++:
SOOP_MAKE_TYPECHECKED_PREDICATE(less, 2, soop::e<int>, soop::e<int>)
```

Darüber hinaus existieren noch zwei Versionen des Makros, die es erlauben dem Prädikat für den Beweiser einen anderen Namen zu geben, was in erster Linie für einige Buildins nötig ist. Gemeinsam mit der Konstante `soop::variadic_rank` die die Überprüfung der korrekten Zahl an Funktionsargumenten deaktiviert, können damit etwa logische Primitive und Gleichheit definiert werden:

```
SOOP_MAKE_RENAMED_PREDICATE(equal, "=", 2)
SOOP_MAKE_PREDICATE(distinct, variadic_rank);
```

Alle drei bisher gezeigten Makros sind nun sind nun über das Allgemeinste vierte Makro `SOOP_MAKE_RENAMED_TYPECHECKED_PREDICATE` definiert, indem sie ihre Argumente an dieses weiterreichen beziehungsweise Vorgaben benutzen die in der Praxis am häufigsten sinnvoll erscheinen: Wird auf einen Namen verzichtet, so wird schlicht der übergebene C++-Bezeichner verwendet, während bei nicht angegebenen Typchecks die Typprüfung deaktiviert wird.

Die Signatur des Makros hat folgende Form:

```
#define SOOP_MAKE_RENAMED_TYPECHECKED_PREDICATE(\
    Identifier, Name, Rank, ...)
```

Hierbei steht `Identifier` für die Basis des C++-Bezeichners des Prädikats, im obigen Beispiel also `equal`, `Name` steht für den Namen den der Beweiser sehen soll (`=`), `Rank` für die Zahl der Argumente und der variadische Teil für die erwarteten Typen, beziehungsweise `void` falls keine Typprüfung stattfinden soll. Im Falle einer Instanziierung erzeugt dieses Makro nun drei Artefakte im Namensraum `preds`: Ein variadisches Klassentemplate `Identifier##_t`, eine Spezialisierung des Selben für den Fall das das einzige Argument `soop::get_meta_information` ist und ein Funktionstemplate `Identifier`, welches eine Instanz des zugehörigen `Identifier##_t` zurück gibt. (`Identifier` ist hier als der Inhalt

des Arguments zu verstehen, während die Doppelraute für die Textkonkatenation des Präprozessors steht; im Beispiel als „`equal_t`“, bzw. „`equal`“.)

Im folgenden nun die Erklärung der einzelnen Teile:

`Identifier##_t` ist das Template, dessen Instanziierung später im statischen Ausdrucksbaum einer Formel erscheint, der Aufbau sieht wie folgt aus:

```
template <typename... Args>
struct Identifier##_t :
        ::soop::basic_predicate<Identifier##_t, Args...>
{
    Identifier##_t(Args... args) :
        ::soop::impl::base_basic_predicate_t<Identifier##_t>{
              std::move(args)...} {}
    static std::string name() { return Name; }
    constexpr static std::size_t rank() { return (Rank); }
    static_assert(
        (sizeof...(Args) == (Rank))
        or ((Rank) == ::soop::variadic_rank),
        "Invalid number of arguments to predicate");
};
```

Die Implementierungen von `name` und `rank` sind trivial und werden daher nicht weiter besprochen.

Das `static_assert` stellt sicher, dass die Zahl der Argumente dem Rang des Prädikats entspricht sofern das Prädikat nicht variadisch ist.

Da `Identifier#_t` `basic_predicate` zur Implementierung benutzt, kann ein Großteil der eigentlichen Logik entfallen, da diese von der Basisklasse bereitgestellt wird.

Als Problematisch hat sich jedoch die Definition des Konstruktors erwiesen: Zwar ist dessen einzige Aufgabe den Basisklassenkonstruktor aufzurufen, allerdings wird dies dadurch erschwert, dass der Name der Basisklasse von der uninstanziierten Form von `Identifier##_t` abhängt, was innerhalb der Klasse kaum ausgedrückt werden kann, da sich der Bezeichner `Identifier#_t` dort implizit auf die instanziierte Form bezieht. Um dieses Problem zu lösen wurde der Templatealias `base_basic_predicate` verfasst, der nichts anderes als ein Alias für die exakte Basisklasse einer gegebene Instanziierung eines von `basic_predicate` erbenden Templates ist. Mithilfe dieses Aliases kann nun der Name der Basisklasse so in Erfahrung gebracht werden, dass der Konstruktor seine Argumente an diese weiterleiten kann.

Die Spezialisierung von `Identifier##_t` stellt nun nichts anderes dar, als eine portable Methode auch ohne konkretes Wissen über die Argumentanforderungen an den Namen und Rang des Prädikats zu gelangen:

```
template <>
struct Identifier##_t<::soop::get_meta_information> {
    static std::string name() { return Name; }
    constexpr static std::size_t rank() { return (Rank); }
};
```

Da die Spezialisierung rein dem Erhalt der Metainformationen dient, erbt sie weder direkt, noch indirekt von `is_predicate` und stellt lediglich Namen und Rang des Prädikats zur Verfügung.

Die eigentliche Erzeugung der Prädikate findet nun über das zuletzt definierte Funktionstemplate `Identifier` statt:

```
template <typename... Args>
auto Identifier(const Args&... args) {
    ::soop::impl::require_valid_types(
        ::soop::impl::actual_types_list<Args...>{},
        ::soop::allowed_types_list<__VA_ARGS__>{});
    return ::soop::make_pred<Identifier##_t>(args...);
}
```

Dieses Template hat zwei Aufgaben: Zum einen die Typprüfung über `require_valid_types`, zum anderen die Inferenz der Argumenttypen. Letztere läuft so ab, dass das Helferfunktionstemplate `make_pred` mit dem Namen des Prädikatentemplates aufgerufen wird und `make_pred` dieses dann mit den Typen der Argumente Instanziiert und eine aus den Argumenten konstruierte Instanz des erzeugten Typen zurück gibt:

```
template <template <typename...> class Pred, typename... Args>
auto make_pred(const Args&... args) {
    return Pred<to_bound_type<Args>...>{args...};
}
```

Die Typüberprüfung findet wiederum so statt, dass `require_valid_types` je eine Instanziierung der variadischen Klassentemplates `actual_types_list` und `expected_types_list` entgegennimmt. Sofern die `actual_types_list` nicht als einziges Typargument `void` enthält, werden nun die zusammengehörenden Typargumente dieser Klassen dahingehend verglichen, ob das tatsächliche Argument zum erwarteten konvertiert werden könnte oder ob es eine Instanziierung von `soop::variable` ist. Ist beides nicht der Fall, wird ein `static_assert` fehlschlagen und damit den Kompiliervorgang zum scheitern bringen.

4.7 Bereitgestellte Prädikate

Standardmäßig stellt SOOP nur ein Minimum an Prädikaten zur Verfügung, von denen die meisten direkt auf in Z3 eingebaute Funktionen abgebildet werden:

- `equal` nimmt zwei Argumente beliebiger Entitäten entgegen und erklärt dem Beweiser, dass beide die selbe Entität repräsentieren.
- `instance_of` nimmt zwei Argumente von denen das erste eine Entität und das zweite ein Typ sein sollte und erklärt, dass das die Entität den angegebenen Typen hat.
- `implies` nimmt zwei Aussagen als Argumente und sagt aus, dass die zweite aus der ersten folgt.
- `distinct` nimmt beliebig Entitäten als Argumente und erklärt dass keine mit irgendeiner der anderen identisch ist.
- `and_` und `or_` nehmen jeweils beliebig viele Argumente und haben die übliche Bedeutung

- `not_` nimmt ein Argument und negiert dessen Wahrheitswert.
- `distinct_range` erhält ein Iteratorpaar über einer Sequenz von Entitäten und erklärt, dass keine zwei Elemente dieser Sequenz identisch sind.
- `forall` nimmt mehrere Variablen über eine `bound_vars`-Instanz als erstes und eine Aussage als zweites Argument und sagt aus, dass die Aussage für alle möglichen Belegungen der übergebenen Variablen gilt.
- `exists` nimmt mehrere Variablen über eine `bound_vars`-Instanz als erstes und eine Aussage als zweites Argument und sagt aus, dass es eine Belegung der übergebenen Variablen gibt, für die die Aussage wahr ist.

5 Anwendung und Evaluation

5.1 Beispielanwendung

Um die Praxistauglichkeit von SOOP zu untersuchen, wurde ein Programm verfasst, dessen Aufgabe es ist Konferenzen zu planen. Die Idee hierbei ist, dass es verschiedene Räume und Zeitslots gibt, in denen von verschiedenen Rednern gehaltene Vorträge stattfinden können. Hierbei kann jeder Redner beliebig viele Vorträge halten und jeder Vortrag von beliebig vielen Rednern gehalten werden.

Das Problem ist insbesondere dadurch interessant, dass es einen praktische Instanz eines NP-vollständigen Problems darstellt, die leicht zu verstehen ist und ein direkt nutzbares Ergebnis enthält. Mit imperativer (oder auch funktionaler) Programmierung lässt sich dieses Problem daher nur mit viel Code befriedigend lösen, da komplizierte Algorithmen benötigt würden um auch nichttriviale Fälle zu lösen. Bei der Verwendung von SOOP, wird dagegen ersichtlich, dass jegliches Entwerfen von komplizierten Algorithmen unnötig wird, da diese bereits in erprobter Form vom Beweiser bereitgestellt werden. Auch wenn diese Herangehensweise in Bezug auf Laufzeit oder Speicherverbrauch weniger effizient sein sollte, als eine auf das Problem zugeschneiderte Lösung, wäre dies vor dem Hintergrund des dramatisch reduzierten Entwicklungsaufwands in vielen Fällen eine valide Kostenabwägung.

Um nun einen Löser zu implementieren, ist es zunächst nötig geeignete Datenstrukturen zu definieren. Diese können etwa wie folgt aussehen:

```
class speaker_t {
public:
    speaker_t(const std::string& name, std::size_t id);
    std::size_t id() const;

private:
    std::string m_name;
    std::size_t m_id;
};
using speaker = soop::e<speaker_t>;
```

Der einzige Unterschied zwischen der Lösung mit SOOP und einer direkten Implementierung ist hier die letzte Zeile und der Name der definierten Klasse: Da alle Entitäten in SOOP von `entity` erben und bestimmte Operationen bereitstellen müssen ist es nötig dies auch für die Redner-Klasse zu tun. Die einfachste Möglichkeit dies zu erreichen ist es, den `e`-Wrapper zu verwenden und diesem einen angenehmen Alias zu geben. Das Resultat hiervon ist Quelltext, der kaum anders aussieht als ohne SOOP.

Ähnlich wie `speaker` wurden auch die Klassen `talk`, `room` und `slot` auf herkömmliche Weise definiert.

Zuletzt wird für diese noch ein Parser implementiert um die relevanten Daten aus einer Textdatei einzulesen. Auch hier ändert sich die Struktur des Codes kaum, lediglich eine Referenz auf eine `soop::ontology` muss zusätzlich an die jeweiligen Funktionen übergeben werden, damit diese in den Konstruktoren der genannten Klassen als erstes Argument verfügbar ist. (Alternativ wäre auch die Verwendung von globalen Variablen *möglich*, wenn auch nicht dem Stand der Softwaretechnik entsprechend). Da der eigentliche Parsercode nicht weiter interessant ist, hier nur die Signatur:

```
struct dataset {
    std::vector<speaker> speakers;
    std::vector<talk> talks;
    std::vector<slot> slots;
    std::vector<room> rooms;
};

dataset read_dataset(std::istream& stream, soop::ontology& o);
```

Um nun eine Lösung für diese Daten zu finden, wird eine Ontologie mit zwei Prädikaten benötigt. Hierfür definieren wir zunächst die Prädikate und deklarieren eine Erzeugerfunktion:

```
// produces an ontology with all the required axioms:
soop::ontology make_ontology();

// The arguments are typechecked in c++:
SOOP_MAKE_TYPECHECKED_PREDICATE(is_speaker_of, 2,
    speaker, talk);
SOOP_MAKE_TYPECHECKED_PREDICATE(talk_assignment, 4,
    talk, speaker, room, slot);
```

In `make_ontolgy` wird eine Ontologie erzeugt, die bereits alle benötigten Typen, Prädikate und Axiome kennt, sodass auf der Verwendungsseite nur noch die konkreten Daten und ihre Beziehungen eingefügt werden müssen:

```
soop::ontology make_ontology() {
    soop::ontology onto{
        soop::type_list<
            talk,
            speaker,
            room,
            slot
        >,
        soop::pred_list<
            preds::is_speaker_of_t,
            preds::talk_assignment_t
        >{}
    };
    onto.add_axiom(uniqueness_of_talks());
    onto.add_axiom(talk_assignment_is_typed());
    onto.add_axiom(only_speakers_hold_their_talk());
    onto.add_axiom(types_are_no_instances());
```

```
    onto.add_axiom(entities_have_only_one_type());
    onto.add_axiom(one_talk_per_speaker_per_slot());
    return onto;
}
```

Bei den Argumenten der `add_axiom`-Aufrufe handelt es sich um Funktionen die je eine `soop::formula` zurückgeben, welche ein formalisiertes Axiom enthält. Exemplarisch sei die Definition von `uniqueness_of_talks` gezeigt:

```
soop::formula uniqueness_of_talks() {
    return forall({t1,t2,s1,s2,r1,r2,sl1,sl2},
        implies(
            and_(
                talk_assignment(t1,s1,r1,sl1),
                talk_assignment(t2,s2,r2,sl2)
            ),
            equal(
                equal(t1,t2),
                and_(equal(r1,r2), equal(sl1,sl2))
            )
        )
    );
}
```

Die lokalen Variablen `t1`, `t2` u.s.w. sind in einem anonymen Namensraum definierte Konstanten deren Typen Instanziierungen des Templates `soop::variable` sind.

Insgesamt benötigt die komplette Axiomatisierung auf diesem Wege deutlich unter 100 Zeilen Quelltext. Eine imperative Implementierung wäre dagegen erheblich länger und komplizierter.

Mit einer auf diesem Wege erzeugten Ontologie `o` und einem Datensatz `data` muss nun noch die Beziehung zwischen verschiedenen konkreten Entitäten deklariert werden:

```
// all speakers participate in all their talks:
for (const auto& talk : data.talks) {
    for (const auto speaker_id: talk->speaker_ids()) {
        o.add_axiom(is_speaker_of(data.speakers.at(speaker_id), talk));
    }
}
// All talks in data.talks are really different talks:
o.add_axiom(distinct_range(data.talks.begin(), data.talks.end()));
```

Nachdem dies getan ist, ist es zum einen trivial nach der Existenz einer Lösung zu fragen (`o.check_sat()`) aber insbesondere ist es möglich, nach einer geeigneten Belegung zu fragen:

```
for (const auto& talk : data.talks) {
    const auto& speaker = data.speakers.at(talk->speaker_ids().front());

    auto solution = o.request_entities<room, slot>(
        talk_assignment(talk, speaker, r, sl), r, sl);
```

```
    const auto& used_room = std::get<0>(solution);
    const auto& used_slot = std::get<1>(solution);

    std::cout
        << talk->title()
        << ": in room #" << used_room->number()
        << ", in slot #" << used_slot->time() << '\n';

    o.add_axiom(talk_assignment(talk, speaker, used_room, used_slot));
}
```

Das abschließende `add_axiom` ist hierbei nötig, um sicherzustellen, dass die Ergebnisse in den nächsten Schleifendurchläufen konsistent bleiben.

5.2 Evaluation

Mit Ausnahme der aus Platzgründen nicht gezeigten weiteren Axiome handelt es sich hierbei um die komplette benötigte Logik zur Lösung des Problems. (Nicht gezeigt wurden auch Dinge wie Fehlerbehandlung und sonstige nicht zur direkten Lösung beitragenden Instruktionen.)

Insgesamt wird das Problem durch die Verwendung von SOOP damit in 265 Zeilen Quelltext gelöst (gezählt von „lizard“[1]), was mit einem direkten Ansatz sicher übertroffen würde. Auch die Komplexität des Quelltexts ist im Verhältnis sehr niedrig: Über 23 Funktionen verteilt lag die von Lizard gemessene durchschnittliche zyklomatische Komplexität (Ein Maß für die Komplexität in Funktionen, das vereinfacht gesagt die Zahl der Pfade durch eine Funktion misst) bei einem Wert von 1,7, wobei lediglich `main` (8) und `read_dataset` (6) einen Wert von 3 überschritten.

Selbst wenn man hierbei die Implementierung von SOOP selbst betrachtet, ändert sich dieses Bild nur wenig: Über 106 Funktionen verteilt beträgt die durchschnittliche zyklomatische Komplexität 1,27 wobei lediglich drei Funktionen einen Wert von vier erreichen. Zum Vergleich sei darauf hingewiesen, dass „lizard“ Funktionen mit einem Wert von unter 16 als unproblematisch erachtet, was von SOOP konsequent stark unterschritten wird.

Auch nach anderen Maßstäben bestehen sowohl beide Codebasen gute Werte: Die durchschnittliche Zahl an Codezeilen pro Funktion beträgt in SOOP vier, mit einem Maximum von 21, während die Beispielapplikation mit einem durchschnittlichen Wert von acht (Maximum: 42, nächstgrößere Funktion: 21) zwar höher, aber immer noch deutlich im überschaubaren Bereich liegt.

Zusammenfassend lässt sich sagen, dass der nötige Aufwand um SOOP in eine existierende Codebasis einzupflegen sehr niedrig ist: Wie im vorherigen Abschnitt zu sehen war, können C++-Datentypen ganz regulär wie für jede andere Herangehensweise definiert werden und dann durch das Hinzufügen einer einzigen Zeile Code für SOOP nutzbar gemacht werden. Im Anschluss ist es zunächst nur nötig, die zugehörige Ontologie allen Konstruktoren zugänglich zu machen, was über globale Ontologien trivial ist, aber auch auf dem saubereren Weg, diese über Argumente durchzureichen, keinen all zu großen Aufwand darstellt.

[1] `https://github.com/terryyin/lizard`

Zuletzt erfolgt die Definition der Axiome. Während diese bei der Verwendung von SOOP explizit aufgeschrieben werden müssen und keine impliziten Annahmen erlaubt sind, ist dies jedoch auch bei allen anderen Ansätzen eine sinnvolle Tätigkeit um logische Fehler leichter zu erkennen. Damit ist diese Arbeit immer in einem gewissen Maße zu aufzuwenden was erneut einen geringen Mehraufwand für die Verwendung von SOOP bedeutet. Die Formulierung der eigentlichen Anfragen ist im Anschluss trivial.

Im Gegensatz hierzu steht der Aufwand einer expliziten Implementierung eines Lösungsalgorithmus einerseits und der traditionellen Verwendung von Ontologien andererseits: Beim expliziten Programmieren wäre eine exakte Axiomatisierung für viele Problemstellungen quasi eine zwingende Vorraussetzung um schnelle und korrekte Algorithmen zu entwerfen. Der Aufwand einer solchen Implementierung wäre in viele Fällen dennoch sehr hoch und die erreichbare Qualität der Lösung insbesondere bei NP-vollständigen Problemen trotzdem oft kritisch, da hier jede gute Lösung eine Heuristik sein muss und es eben im allgemeinen Fall immer nötig ist, den Großteil des kompletten Lösungsraums zu durchsuchen.

Die klassische Verwendung von Ontologien hätte andere, aber nicht geringere Nachteile. Prinzipiell gibt es hier zwei grundlegende Verfahren: Die komplett manuelle Übersetzung und die Verwendung klassischer Ontologien-APIs. Mit manueller Übersetzung ist hierbei gemeint, dass die Daten von einem Menschen gelesen und tatsächlich von Hand über einen Ontologie-Editor benutzbar gemacht werden. Offensichtlich skaliert ein solches Vorgehen sehr schlecht und wird auch nie die Übersetzungsgeschwindigkeit einer Maschine erreichen.

Die maschinelle Übersetzung etwa mit den in Kapitel 2 vorgestellten Bibliotheken stellt sich daher als für viele Fälle überlegene Lösung dar, teilt jedoch einige fundamentale Probleme: Für alle Entitäten werden in beiden Herangehensweisen Objekte erzeugt die für bereits existierende Datenobjekte (also Objekte der fraglichen Programmiersprache die weitere, für die Ontologie potentiell nicht relevante Informationen enthalten) stehen und die zugehörigen Individuen in der Ontologie repräsentieren. In der Folge ist eine händisch zu programmierende Übersetzung zwischen diesen Objektarten notwendig, was einen erheblich höheren Aufwand darstellt als die implizite Verwaltung über SOOP.

Trotz dieses Mehraufwands bringt die klassische Verwendung von Ontologien kaum einen Vorteil, da der Hauptaufwand bei der Verwendung von SOOP in der Definition der Axiome liegt, was mit allen anderen Ansätzen ebenfalls zwingend nötig ist. Das Fazit lautet daher, dass SOOP auf diesem Gebiet große Vorteile bringen kann, ohne bedeutende Nachteile, wie etwa einen erhöhte Entwicklungsaufwand oder starke Performance-Einbußen zu haben.

6 Diskussion und Ausblick

6.1 Diskussion

Die motivierende Idee der Arbeit war, statt Dinge in OOP auszuprogrammieren und beispielsweise Referenzen in Objekten zu speichern, derartige Beziehungen in einer Ontologie auszudrücken. Hierdurch sollte es auch ermöglicht werden, die Interoperabilität zwischen verschiedenen Bibliotheken zu erhöhen indem standardisierte Ontologien wie SUMO verwendet werden.

Im Verlauf der Arbeit verschob sich der Fokus in die Richtung des nun vorliegenden Endergebnisses die Ontologie aus den Daten eines regulären Programms zu erzeugen. Für die ursprünglichen Ziele bedeutete dies, dass sie teilweise fallen gelassen, teilweise aber auch deutlich erreicht wurden. So ist die semantische Verbindung verschiedener Objekte problemlos möglich und auch komplizierte Abfragen können leicht gestellt werden. Um etwa zu signalisieren, dass ein Reifen Teil eines bestimmten Fahrzeuges ist, kann man mit SOOP folgendes Schreiben:

```
onto.add_axiom(is_tire_of(tire, vehicle));
```

Je nach Definition von `is_tire_of` können auf diesem Wege auch verschiedene Bibliotheken verbunden werden, die sonst nichts voneinander wissen. Auch wenn `vehicle` in diesem Beispiel aus einer Bibliothek stammt, in der noch nicht einmal das Konzept von Reifen bekannt ist, kann diese Information nun dargestellt werden. Auch ist es möglich nun Anfragen zu stellen ob ein bestimmtes Fahrzeug einen kaputten Reifen hat:

```
onto.request(exists{t},
    and_(is_tire_of(t, vehicle), is_broken(t)));
```

Oder, wenn bekannt ist, dass dies der Fall ist, zu fragen welcher Reifen kaputt ist:

```
const auto broken_tire = std::get<0>(onto.get_entities<tire>(
    and_(is_tire_of(t, vehicle), is_broken(t)), t));
```

Offensichtlich wurde hiermit das Ziel der semantischen Verbindung von Objekten ohne zwingende OOP-Beziehung erreicht. Auch komplexere Anfragen als die Gezeigte sind möglich.

Die Verbindung über eine standardisierte Ontologie wie SUMO wurde zwar nicht realisiert, könnte aber über eine ontologienspezifische Bibliothek getan werden. Hierfür würde man einfach alle Prädikate der fraglichen Ontologie als SOOP-Prädikate und alle Individuen als reguläre SOOP-Objekte implementieren.

Die Hauptproblematik für SUMO konkret stellt dessen Anforderung dar, Zugriff auf Prädikatenlogik höherer Ordnung zu haben, was weder von Z3, noch SPASS bereitgestellt wurde und aus diesem Grund auch in SOOP nicht verfügbar ist. Mit einem geeigneten Beweiser wäre eine entsprechende Erweiterungen von SOOP jedoch möglich.

Das modifizierte Ziel, Semantik von regulären Daten die als herkömmliche Objekte in einem Programm vorliegen in eine Ontologie zu übertragen wurde ebenfalls erreicht.

6.2 Ausblick

Im Rahmen der Arbeit wurden viele Gebiete nur angeschnitten, aber (meist aus zeitlichen Gründen) nicht weiter verfolgt. Viele der im Folgenden genannten Konzepte ließen sich relativ leicht implementieren, auch wenn gelegentlich eine ausführliche Diskussion der Vor- und Nachteile verschiedener Möglichkeiten zwingend erforderlich wäre.

Garbage-Collection Eine wichtige Optimierung wäre automatische Ressourcenbereinigung für Entitäten und Axiome in einer Ontologie: Aktuell werden alle Entitäten in einem großen `std::vector` gespeichert und belegen auch nachdem sie entfernt wurden Platz. Dies stellt insbesondere bei Ontologien in denen häufig Objekte hinzugefügt und entfernt werden ein großes Problem dar, welches sich im Prinzip einfach lösen ließe:

Durch das Hinzufügen eines Zählers für die tatsächlich enthaltenen Element ließe sich das Verhältnis von benutzten Entitäten zu benutztem Platz berechnen sodass beim Unterschreiten einer Grenze (etwa 0.5) automatisch eine Kompaktierung stattfindet. Die einfachste Möglichkeit hierfür wäre den Vector mit `std::remove` zu kompaktieren und im Anschluss alle Entitäts-IDs auf die neuen Indices umzustellen, was auch über den in der Entität gespeicherten Zeiger auf `const` funktionieren könnte, indem die ID `mutable` gespeichert wird. (Aus einem ähnlichen Grund ist bereits der Ontologie-Zeiger `mutable`.) Im Anschluss müssten noch die Entitäts-IDs der Axiome angepasst werden, was jedoch ebenfalls kein all zu großes Problem darstellen sollte.

Weiterverwendung des Beweiserzustandes Eine andere entscheidende Optimierung wäre die Weiterverwendung des Beweiserzustandes nach einer Anfrage. Bei dem im letzten Kapitel vorgestellten Anwendungsbeispiel des Konferenzplaners ist es etwa nötig, für jeden Vortrag eine neue Anfrage zu stellen, die das gesamte Problem von Grund auf neu löst, obwohl bereits im vorherigen Schritt eine vollständige Lösung gefunden wurde. Hier wäre es sehr wünschenswert, den Zustand nach der ersten Anfrage für weitere Anfragen zu verwenden.

Ein möglicher Ansatz hierfür wäre ein Anfrage-Objekt zu erzeugen welches den Zustand des Beweisers erst nach seiner Zerstörung durch den Destruktor zurücksetzt und über das Anfragen gestellt werden können. Eine optimale Lösung hierfür sollte sowohl konstante Anfragen ermöglichen, als auch das Hinzufügen weiterer Axiome erlauben. Für Letzteres wäre ein vielversprechender Ansatz, dem Objekt eine Methode zu geben, die ein neues derartiges Objekt zurück gibt, so dass diese weiteren Axiome nur für dessen Lebensdauer gelten und in der dessen Vorfahren nicht benutzt werden dürfen.

Das folgende Beispiel skizziert wie ein solcher Ansatz in der Praxis aussehen könnte:

```
auto questioner_1 = onto.prepare_request();
std::cout << questioner_1.request(...);
std::cout << questioner_1.get_entity<...>(...);
{
```

```
    auto questioner_2 = questioner_1.with_further_axiom(...);
    // questioner_1 must not be used here
    std::cout << questioner_2.request(...);
}
// questioner_1 can be used again
```

Neben dem kleinen Problem, dass der Programmierer hier selbst auf die Lebensdauern der Anfrageobjekte achten muss, stellt die potentiell länger anhaltende Blockade des Beweisers ein Problem für die aktuelle Implementierung dar: Gegenwärtig existiert pro Thread höchstens ein Beweiserprozess, der von allen anfragenden Ontologien gleichzeitig benutzt wird. Beim Vorhandensein mehrerer Ontologien wäre eine solche Blockade praktisch untragbar, so dass andere Lösungen gefunden werden müssten. Möglichkeiten wären hier, einen Prozess pro Ontologie zu verwenden (wodurch zwar immer noch verschiedene Anfragen an die selbe Ontologie blockiert werden könnten, dass praktische Problem jedoch sehr viel kleiner wäre), einen Prozess pro Anfrage zu verwenden (was sehr teuer wäre) oder eine komplizierte Prozessverwaltung, bei der ähnlich wie im zweiten Ansatz neue Prozesse erzeugt werden, jedoch nur, sofern aktuell alle blockiert wären.

Kopien Eine Problematik die nicht technischer Natur ist, ergab sich bei Kopien von Entitäten. Prinzipiell sind hier drei Ansätze naheliegend:

- Eine Kopie einer Entität ist eine neue Entität, die keine Eigenschaften von ihrem Original erbt.
- Eine Kopie ist das Selbe wie das Original, der Ontologie wird mitgeteilt, dass das alte und das neue Objekt identisch sind.
- Eine Kopie hat die selbe Semantik wie das Original, alle Axiome die Aussagen über das Original treffen werden (tief) kopiert und alle Erwähnungen des Originals durch Erwähnungen des neuen Objekts ersetzt.

Für jede dieser drei Möglichkeiten lassen sich Beispiele finden in denen sie angemessen sind und Beispiele, in denen sie es nicht sind. Nicht zuletzt die Verwendung von Puffern als Optimierung machen eine Aussage was sinnvoll ist selbst für konkrete Typen im Allgemeinen fast unmöglich. Eine ausgiebige Untersuchung verschiedener Fälle und Heuristiken erscheint als vielversprechendes Thema für weitere Untersuchungen.

Aufgrund der hohen Komplexität des Themas, sind Entitäten in SOOP standardmäßig nicht kopierbar, es ist jedoch möglich für selbstdefinierte Entitäten einen Kopierkonstruktor zu erzeugen, der zunächst eine freie Entität hervorbringt und diese dann mit der gewünschten Semantik zur fraglichen Ontologie hinzufügt.

Entitätssequenzen Eine weitere Thematik, die wieder mehr in Richtung Optimierung, aber auch Bedienkomfort geht ist die bessere Unterstützung von Entitätssequenzen: Um Axiome über alle Elemente einer Sequenz zu erzeugen, ist es gegenwärtig nötig, dies komplett von Hand mit einer von `is_predicate` erbenden Klasse und zugehörigen Funktionen zu implementieren. Eine Verbesserung dieser Situation wäre sicherlich wünschenswert, auch wenn SOOP gegenwärtig die Erzeugung variadischer Prädikate in Z3 nicht unterstützt.

Verwandt mit dieser Thematik ist die Anfrage einer Sequenz von Werten dynamischer Länge. welche aktuell noch keinerlei Unterstützung erfährt. Die Verfügbarkeit eines derartigen

Mechanismus würde auch das Bedürfnis nach der Weiterverwendung des Beweiserzustandes (wie oben beschrieben) reduzieren.

Inferenz der Prädikatentypisierung Eine weitere Verbesserung der Benutzbarkeit wäre bei der Typisierung von Axiomen nötig: Aktuell muss der Ontologie über ein vom Benutzer zu verfassendes Axiom mitgeteilt werden, welche Anforderungen Prädikate an die Typen ihrer Argumente stellen, was zum Beispiel so aussehen kann:

```
onto.add_axiom(forall({s, t}, implies(
    is_speaker_of(s, t), and_(
        instance_of(s, type<speaker>),
        instance_of(t, type<talk>)))));
```

In Worten besagt dieses Axiom, dass jede erfüllende Belegung eine Instanz des Typs `speaker` als erstes und eine des Typs `talk` als zweites Argument erhält. Die idiomatische Definition von `is_speaker_of` mit SOOP sieht wie folgt aus:

```
SOOP_MAKE_TYPECHECKED_PREDICATE(is_speaker_of, 2, speaker, talk)
```

Offensichtlich ist damit die Typisierung redundant vorhanden und ließe sich auch statisch in den Typ des Prädikats auf eine Weise kodieren, dass ein entsprechendes Axiom beim Hinzufügen eines Prädikats zu einer Ontologie automatisch angelegt wird.

Inferenz der benutzten Variablen Auch im Kontext von Variablen ließe sich das Typsystem für mehr Komfort und Sicherheit verwenden. Das Hauptproblem ist hierbei im Moment, dass instanziierte Prädikate nicht speichern, welche Variablen in ihnen verwendet wurden. Dies ließe sich insbesondere dadurch leicht ändern, dass Prädikate eine rekrusiv erzeugte Liste der in ihnen benutzten Variablen in ihrem Typ kodieren, was eine Vielzahl von Verbesserungen erlauben würde:

- Detektion verdeckender Variablennamen, welche in der Praxis meist Fehler sein dürften. Im folgenden Beispiel ist es zum Beispiel eher unwahrscheinlich, dass `bar` zweimal mit dem selben Argument aufgerufen werden sollte, aber der Ausdruck an sich ist fehlerfrei:

  ```
  forall({x}, and_(foo(x), forall({x}, bar(x, x))))
  ```

- Detektion unsauber definierter Variablen. Aktuell ist es möglich dass eine Variable in C++ unter mehreren Namen auftaucht, was zu sehr verwirrenden Laufzeitproblemen führen kann:

  ```
  soop::variable<'x', '1'> x1;
  soop::variable<'x', '1'> x2;
  ```

  Hierbei handelt es sich nicht um ein hypothetisches Problem: Als Resultat eines klassischen Copy-Paste-Fehlers trat exakt diese Art von Bug initial bei der Entwicklung der Beispielapplikation auf und könnte ebenfalls durch das Verbot mehrerer namensgleicher Variablen abgefangen werden.

- Detektion ungebundener Variablen: Quantoren würden in einer guten Implementierung die von ihnen definierten Variablen als gebunden markieren, was in einer Liste der in der Formel verwendeten ungebundenen Variablen resultieren könnte. Diese Liste ließe sich für verschiedene Zwecke verwenden, die naheliegendsten wären hierbei die folgenden:
  - Überprüfung ob die Liste leer ist um die ordentliche Bindung aller Variablen zu prüfen.
  - Die impliziete Generierung eines Allquantors der diese Variablen definiert; dies würde dem Ansatz von SUMO ähneln und in vielen Fällen kürzere Axiome erlauben.
  - Ebenfalls möglich wären Existenzquantoren, auch wenn der Nutzen hiervon eher fraglich wäre.
  - Bei Anfragen könnten die Lösungsvariablen inferiert werden, was deren explizite Nennung unnötig machen würde; die Gefahr hierbei wäre jedoch, dass eine unbeabsichtigte Vertauschung der Reihenfolge weniger deutlich zu erkennen wäre.

Typisierte Variablen Auch ansonsten gibt es im Kontext von Variablen noch diverse Verbesserungsmöglichkeiten: Gegenwärtig sind Variablen etwa von der Typüberprüfung bei Prädikaten ausgenommen, da sie nie typisiert sind. Dies ist jedoch nicht aufgrund einer fundamentalen Beschränkung der Fall. Typisierte Variablen sind prinzipiell durchaus denkbar und würden SOOP wohl in vielfacher Weise bereichern, etwa indem die Typen automatisch als Anforderung in Formeln eingefügt werden, was potentiell sogar eine höhere Beweisgeschwindigkeit durch bessere Restriktionen zur Folge haben könnte.

Typisierung verschiedener Variablenarten Im Kontext einer solchen Typisierung wäre es auch wünschenswert Lösungsvariablen von „Quantorvariablen“ statisch unterscheidbar zu machen um die Gefahr einer Namenskollision weiter zu reduzieren und etwa bei der oben vorgeschlagenen Inferenz der ungebundenen Variablen je nach Variablentyp verschieden zu verfahren. Eine einfache Unterscheidungsmöglichkeit könnte dadurch erzeugt werden, dass Variablentypen nur noch über zwei verschiedene Aliastemplates erzeugt werden können die dem (durch die Templateargumente übergebenen) Variablennamen je nach Art ein verschiedenes Präfix geben.

Nutzung des Z3-Typsystems Ein weiteres Gebiet auf dem noch Untersuchungen stattfinden könnten ist die Verwendung des von Z3 selbst bereit gestellten Typsystems. Aktuell werden alle Entitäten als Instanzen ein und des selben Typs dargestellt, womit sich SOOP an SUMOs `Entity` orientiert. Inwieweit hier eine stärke Ausnutzung der von Z3 zur Verfügung gestellten Features sinnvoll wäre, wurde bisher nicht untersucht.

Vereinigung verschiedener Entitätenarten Zu guter Letzt sei auch noch die aktuelle Trennung von Typ- und Laufzeitentitäten in SOOP angesprochen. Diese erscheint zwar zunächst vor dem Hintergrund klassischer Verwendungsmuster naheliegend, aber eine detaillierte Untersuchung ob diese Abweichung, zwischen dem was in C++ und Z3 geschieht, die Vorteile rechtfertigt, existiert zum jetzigen Zeitpunkt nicht.

7 Glossar

Atom (prädikatenlogisch) siehe Entität

Axiom Hier eine prädikatenlogische Aussage die als wahr definiert wird.

CRTP Curiously Recurring Template Pattern. Eine Technik um nichtvirtuelle Methoden aus einer Basisklasse aufzurufen

Entität Ein konkretes Objekt über das Aussagen gemacht werden können.

Individuum Siehe Entität.

Klasse Eine Definition des Aufbaus von C++-Objekten.

Laufzeitentität Ein konkretes C++-Objekt das als Entität benutzt wird. Der Typ einer Laufzeitentität ist immer eine Typentität.

Objekt Eine Instanz einer C++-Klasse.

Prädikat Ein benanntes Tupel das Individuen und Variablen in Beziehung setzt.

Quantor Allquantor($\forall$) und Existenzquantor($\exists$). Ermöglichen prädikatenlogische Aussagen über Existenz und Allgemeingültigkeit.

RTTI Runtime Type Information. Ermöglicht abfrage von Typinformationen zur Laufzeit.

Typentität Ein C++-Typ der als Entität benutzt wird. Instanzen des Typs sind meist Laufzeitentitäten.

Variable (prädikatenlogisch) Ein an einen Quantor gebundener lokaler Bezeichner der für alle Individuen stehen kann.

8 Literaturverzeichnis

[1] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[2] Andreas Eberhart and Sudhir Agarwal. Smartapi - associating ontologies and apis for rapid application development. In *Ontologien in der und für die Softwaretechnik*, März 2004.

[3] Thomas R. Gruber. A translation approach to portable ontology specifications. 1992.

[4] Matthew Horridge and Sean Bechhofer. The owl api: A java api for owl ontologies. *Semant. web*, 2(1):11–21, January 2011.

[5] Apache Jena. Apache jena. *jena. apache. org [Online]. Available: http://jena. apache. org [Accessed: Mar. 20, 2014]*, 2013.

[6] Holger Knublauch, Ray W. Fergerson, Natalya F. Noy, and Mark A. Musen. The protégé owl plugin: An open development environment for semantic web applications. pages 229–243. Springer, 2004.

[7] Seiji Koide, Jans Aasman, and Steve Haflich. Owl vs. object oriented programming. In *the 4th International Semantic Web Conference (ISWC 2005), Workshop on Semantic Web Enabled Software Engineering (SWESE)*. Citeseer, 2005.

[8] Eyal Oren, Renaud Delbru, Sebastian Gerke, Armin Haller, and Stefan Decker. Activerdf: Object-oriented semantic web programming. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 817–824, New York, NY, USA, 2007. ACM.

[9] Adam Pease. *Ontology: A Practical Guide*. Articulate Software Press, Angwin, CA, 2011.

[10] Colin Puleston, Bijan Parsia, James Cunningham, and Alan Rector. *The Semantic Web - ISWC 2008: 7th International Semantic Web Conference, ISWC 2008, Karlsruhe, Germany, October 26-30, 2008. Proceedings*, chapter Integrating Object-Oriented and Ontological Representations: A Case Study in Java and OWL, pages 130–145. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[11] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. Spass version 3.5. In *Automated Deduction–CADE-22*, pages 140–145. Springer, 2009.