

Schema Integration Based on Structure and Behaviour



Christiaan Thieme

Schema Integration Based on Structure and Behaviour

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam,
op gezag van de Rector Magnificus
prof. dr. P.W.M. de Meijer
ten overstaan van
een door het college van dekanen ingestelde commissie
in het openbaar te verdedigen
in de Aula der Universiteit
op dinsdag 23 mei 1995 te 14.00 uur

door
Christiaan Johannes Emiel Thieme
geboren te Tegelen

Promotor: prof. dr. M.L. Kersten
Co-promotor: dr. A.P.J.M. Siebes
Faculteit: Wiskunde en Informatica

Woord vooraf

Bloed, zweet en tranen.

Dat heeft de totstandkoming van dit proefschrift mij gekost. Dit klinkt waarschijnlijk erger dan het is. Ik heb veel mooie momenten meegemaakt en veel geleerd op allerlei gebieden. Het proefschrift was er echter nooit gekomen zonder de hulp van collega's, vrienden en familie. Een aantal van hen wil ik speciaal noemen.

Als eersten wil ik Martin Kersten, mijn promotor, en Arno Siebes, mijn copromotor, bedanken dat zij mij een positie hebben aangeboden in de Onderzoeksgroep Databanken van het CWI, waar ik in alle vrijheid mijn onderzoek heb kunnen doen. Bijzondere waardering heb ik voor Arno Siebes, die altijd klaar stond om als klankbord te fungeren of een 'peptalk' te geven. De soms pittige discussies over mijn onderzoek wisselden af met prettige gesprekken achter een glas bier.

Ik bedank Peter Apers, Peter van Emde Boas, Gregor Engels, Paul Klint, professor Maes en professor Saltor dat zij zitting hebben genomen in mijn promotiecommissie en dit proefschrift hebben beoordeeld. I'm honoured that Felix Saltor of the Universitat Politècnica de Catalunya agreed to be a committee member and refereed this thesis.

Het was Arno Siebes die mij in contact bracht met AXIS, een netwerk van promovendi op het gebied van de informatiesystemen. De talloze discussies die we hadden tijdens onze bijeenkomsten zijn voor mij een bron van inspiratie geweest.

Met plezier denk ik terug aan de lunches, waar leden en ereleden van de Afdeling AA hun werk even vergaten om zich te storten in allerlei maatschappelijke en populair-wetenschappelijke discussies. Ik dank hen allemaal voor de prettige sfeer die er heerste tijdens en ook buiten de lunches.

Speciaal wil ik Leonie van der Voort en Carel van den Berg bedanken, die gedurende een aantal jaren mijn kamergenoten waren. Bij hen kon ik terecht met vragen over software, voor discussies over databanken of gewoon voor 'social talk'. Ook wil ik Joke Sterringa, met wie ik opgezette tijden kon bijkletsen, bedanken voor haar warme belangstelling.

Ik bedank Marja Hegt en Mieke Bruné, die met hun deskundige ondersteuning het leven van de onhandige wetenschapper weten te verlichten.

Tenslotte wil ik mijn vrienden en familie bedanken. Zonder de liefdevolle steun van Hans, Martha, Robin en Ronald was het me nooit gelukt. Ook de verbondenheid met Harald, Ita en Cal is voor mij zeer belangrijk geweest.

Speciale dank geldt mijn ouders, die me geleerd hebben verantwoordelijkheid te dragen en me gesteund hebben in al mijn keuzen.

Contents

1	Introduction	1
1.1	Integration of database systems	1
1.2	Schema integration	5
1.2.1	Preintegration and comparison	6
1.2.2	Conforming and restructuring	8
1.3	This thesis	9
1.3.1	Outline	10
2	An introductory example	11
2.1	Comparison	17
2.2	Conforming	20
2.3	Merging and restructuring	22
I		25
3	Database schemas	27
3.1	Syntax	28
3.2	Underlying types	29
3.3	Functional forms	33
4	Properties of underlying types	39
4.1	Syntax and semantics	40
4.2	Type equivalence	46
4.2.1	Derivation system and normalisation process	47
4.2.2	Soundness w.r.t. structural equivalence	51
4.2.3	Completeness w.r.t. structural equivalence	56
4.2.4	Equivalence of structural and extensional equivalence	60
4.2.5	Equivalence of derivable and reducible equivalence	68
4.2.6	Decidability of derivable equivalence	69
4.3	Subtyping	70

4.3.1	Derivation system	71
4.3.2	Soundness w.r.t structural subtyping	73
4.3.3	Completeness w.r.t. structural subtyping	77
4.3.4	Equivalence of structural and extensional subtyping	79
4.3.5	Decidability of derivable subtyping	80
5	Properties of functional forms	81
5.1	Syntax and semantics	81
5.2	Decidability of equality	86
5.2.1	Decision procedure	92
6	Schema comparison	95
6.1	Comparison of attributes	96
6.2	Comparison of methods	99
7	Schema integration	105
7.1	Integration of attributes	105
7.2	Integration of methods	107
7.3	Integration of class hierarchies	108
7.4	Application	110
II		113
8	Extended database schemas	115
8.1	Syntax	115
8.2	Inheritance of attributes	117
8.3	Underlying constraints	120
8.4	Inheritance of methods	122
9	Schema transformations	131
9.1	Type transformations	131
9.2	Transformations on predicates and functions	136
10	Properties of schema transformations	143
10.1	Normal forms	143
10.2	Soundness	146
10.3	Completeness	150
11	Combined approach	157
11.1	Comparison of classes	157
11.2	Integration of classes	160
11.3	Application of schema transformations	161

12 Conclusion	173
12.1 Summary	173
12.2 Evaluation	175
12.3 Further research	176
A Complexity of subclass relation	179
Bibliography	183
Samenvatting	189

Chapter 1

Introduction

Cooperation between information systems has become a vital issue for both commercial and non-commercial organisations. In the last two decades, a lot of information systems have been developed, ranging from information systems for student administration and banking applications to information systems for office automation, computer-aided design and manufacturing, scientific visualisation, and hypermedia applications. The purpose of these systems is to support the processes in an organisation, which aim at achieving its goals.

Since the processes in an organisation change, due to changes in the environment or in the organisation (e.g., the goals can change), its information systems have to change as well. An important change is that different processes have to cooperate more and more, both within an organisation and between organisations. As a consequence, it no longer suffices to have isolated information systems for different organisational processes. Instead, systems have to cooperate in one way or another.

For example, the introduction of a human resource management programme can lead to a management information system that requires information from the systems in the personnel department and the sales department. Similarly, the merger of two banking corporations will result in an integrated transaction system and the cooperation between the police forces of different countries will require the exchange of information between the different police information systems.

1.1 Integration of database systems

Cooperation between information systems requires ‘integration’ of the database subsystems that contain the user data, i.e., the user-defined functions, types, and type instances. Integration of database systems can be achieved in several ways. To illustrate this, a simple architecture for database systems is given in Figure 1.1.

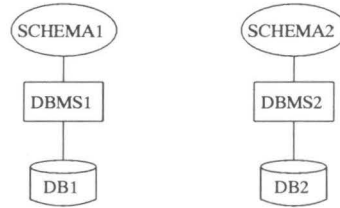


Figure 1.1

The schema (e.g., SCHEMA1) contains the types and functions of the database system, and the database (e.g., DB1) contains the type instances. The database management system (e.g., DBMS1) handles user requests to query and manipulate the schema and the databases. Now, suppose we want to integrate the database systems in Figure 1.1. The integration process depends on whether the database management systems (DBMS1 and DBMS2) are equal or not. If they are, the integration process is referred to as homogeneous database system integration, otherwise it is referred to as heterogeneous database system integration.

We start with homogeneous database system integration. The integrated database system can be characterised by its type: a single-database system (abbreviated as SDBS), managed by one database management system, or a multi-database system (abbreviated as MDBS), managed by two or more database management systems. An SDBS will be chosen if it is possible to centralise the management of the data. The options for an SDBS are shown in Figure 1.2, where DBMS is the common database management system.

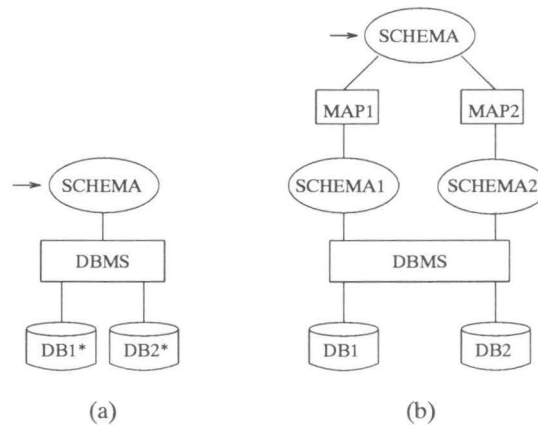


Figure 1.2

Figure 1.2(a) shows the tightly-integrated SDBS option, which is used if the local databases do not have to remain unchanged. SCHEMA is obtained by integrating SCHEMA1 and SCHEMA2, and DB1* and DB2* are obtained by translating DB1 and DB2 into instances of SCHEMA. Once the original databases have been

translated, all queries and updates against the integrated schema can directly be applied to the translated databases. Figure 1.2(b) shows the loosely-integrated SDBS option, which is used if the local databases have to remain unchanged; for example, because the cost of data conversion is too high. The local schemas (SCHEMA1 and SCHEMA2) are integrated, but the databases (DB1 and DB2) are left unchanged. Mappings (MAP1 and MAP2) are used to translate instances of SCHEMA1 and SCHEMA2 into instances of SCHEMA. Only queries and updates against the integrated schema are allowed and will be translated into queries and updates against the local schemas. Furthermore, the results of a query against the local schemas have to be combined into one result for the integrated schema using the mappings.

An MDBS will be chosen if it is not possible to centralise the management of the data. The options for an integrated MDBS are shown in Figure 1.3.

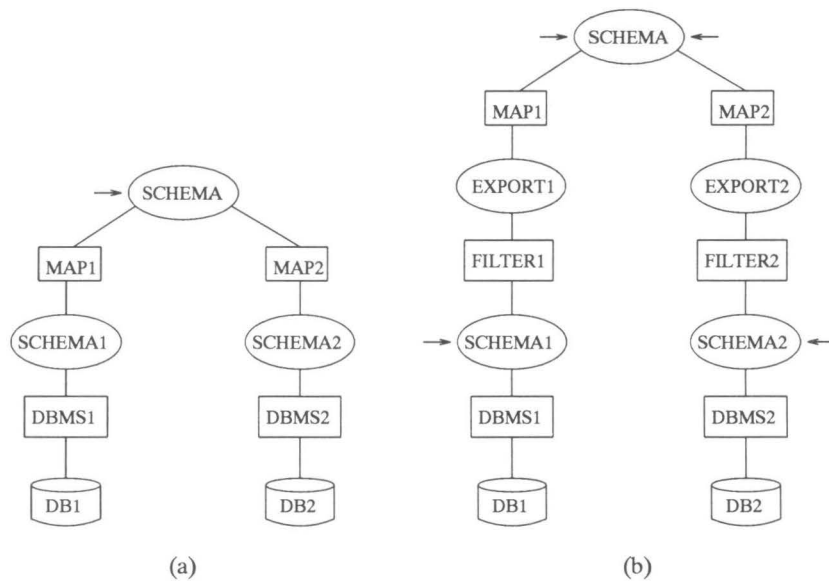


Figure 1.3

Figure 1.3(a) shows the globally-autonomous MDBS option, which is used if the local database systems do not have to remain autonomous; for instance, if two banking corporations want an integrated transaction system. The globally-autonomous MDBS option is the same as the loosely-integrated SDBS option, except for the separate management of the two databases. Figure 1.3(b) shows the locally-autonomous MDBS option (also called federated database system), which is used if the local database systems have to remain autonomous; for example, if the police forces of different countries want to exchange information between the different police information systems. For this option, the adminis-

trators of the database systems have to define export schemas (EXPORT1 and EXPORT2) representing the parts of the local schemas which are available to other database systems. Filters (FILTER1 and FILTER2) are used to translate instances of the local schemas into instances of the export schemas. The federated schema (SCHEMA) is obtained by integrating EXPORT1 and EXPORT2. Again, mappings are used to translate instances of EXPORT1 and EXPORT2 into instances of SCHEMA. Queries and updates against the local schemas are still allowed, but queries against the federated schema are also allowed. Such a query will be translated into queries against the export schemas and then into queries against the local schemas. Furthermore, the results of the queries against the local schemas have to be combined into one result for the integrated schema using the mappings and the filters. For more details on federated database systems, see [48] and [26].

Heterogeneous database system integration, where DBMS1 is not equal to DBMS2, is similar to the homogeneous case, except for an additional translation step. To obtain an integrated SDBS, first a common DBMS is chosen and the local schemas are translated into the data definition language of the common DBMS and the databases are translated into instances of the translated schemas. Second, one of the options for the homogeneous case (see Figure 1.2) is applied to the translated database systems. To obtain an integrated MDBS, first a common data definition language is chosen and the database systems are extended as in Figure 1.4.

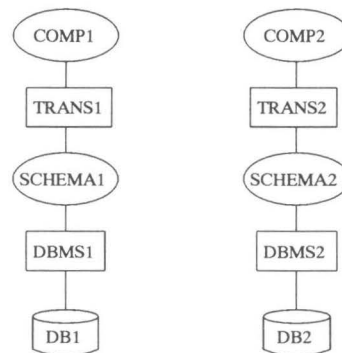


Figure 1.4

The component schemas (COMP1 and COMP2) are obtained by translating the local schemas into the common data definition language. Transformations (TRANS1 and TRANS2) are used to translate instances of the local schemas into instances of the component schemas. Second, one of the options for the homogeneous case (see Figure 1.3) is applied to the extended database systems.

1.2 Schema integration

This thesis focuses on schema integration: defining the integrated schema with respect to a number of component schemas and defining the mappings between the component schemas and the integrated schema. Besides being important for integration of existing database systems, schema integration is also important in traditional database design, when several user views, i.e., schemas developed for different user groups, have to be unified into a global schema. As argued in [8], schema integration is a difficult task, because the integrated schema must satisfy a number of properties:

1. *Correctness.* The integrated schema must contain all concepts present in any component schema correctly.
2. *Minimality.* If the same concept is represented in more than one component schema, it must be represented only once in the integrated schema.
3. *Understandability.* The integrated schema must be easy to understand for the designer and the end user.

The second property requires that different representations of the same concept can be identified. This is the basic problem of schema integration and the main reason why schema integration is a non-trivial task.

In [8], a framework for comparing integration methods is given that identifies four integration phases:

1. *Preintegration.* In the first phase, an integration policy is chosen and additional information, such as assertions between component schemas, is gathered. An integration policy deals with choosing an order of integration (if there are more than two schemas) and assigning preferences to portions of schemas. Assertions are used to define correspondences between and constraints on attributes and object types in different schemas.
2. *Comparison.* Second, the component schemas are compared to determine structural and semantic similarities between attributes and object types. Furthermore, the schemas are analysed to detect naming conflicts, such as the same name for different concepts, and structural conflicts, such as different types for the same concept.
3. *Conforming.* In the conforming phase, the conflicts found in the comparison phase have to be resolved to enable merging of the schemas. Conflicts are resolved in close interaction with the designer by applying renaming operations, type transformations, and other transformations.
4. *Merging and restructuring.* Finally, the schemas are merged using superimposition and the resulting schema is analysed and restructured to achieve minimality and understandability.

A lot of integration methods have been proposed in the literature, but only some of them cover the integration process completely. In the following two subsections, we compare methods that cover at least one of the last three phases. First, we focus on the preintegration and comparison phases, and second, we focus on the conforming and restructuring phases.

1.2.1 Preintegration and comparison

We can distinguish between several types of integration methods based on the level of support for detecting similarities. Some integration methods do not support detection of similarities, but ask the designer to define correspondences between attributes, entity types, and relationship types:

Biskup and Convent [9] and Casanova and Vidal [14] use dependencies to define equality, containment, overlap, and disjointness of attributes.

Elmasri and Navathe [18, 42, 40] use assertions on equality, containment, overlap, and disjointness of domains of classes.

Mannino, Navathe, and Effelsberg [36, 37] use assertions between attributes and assertions between entity types.

Spaccapietra, Parent, and Dupont [54, 56, 55] use assertions between elements (attributes and object types), between attributes of corresponding object types, and between paths that connect elements.

Gangopadhyay and Barsalou [21] use assertions in second-order logic to describe discrepancies in data semantics and differences in data structures.

Other integration methods ask the designer to give semantic information on attributes, entity types, and relationship types:

Siegel and Madnick [53] use semantic rules to interpret attributes in a semantic domain.

Sheth, Gala, and Navathe [46, 47] and Yu, Sun, Dao, and Keirse [65] define correspondence between attributes and semantic points or concept vectors to define the meaning of attributes.

Shoval and Zone [52] use assertions that define the meaning of entity types and relationship types.

Semantic information on attributes, entity types, and relationship types can be established for a lot of candidate schemas in advance, in a ‘general preintegration phase’. In fact, a number of methods use such a general preintegration phase to support detection of similarities:

Urban and Wu [63] use mappings between the local and global data model to define properties of local schemas.

Bright and Hurst [11] use a taxonomy of terms based on a dictionary or thesaurus of the English language to define relatedness of terms.

Yu, Jia, Sun, and Dao [64] use a knowledge base containing concepts, relationships between concepts, and a list of keywords to map names to a set of concepts.

Fankhauser, Kracker, and Neuhold [20] use a terminological database containing probabilistic information on associations of terms and on specialisation of terms to derive relevance ratios of attributes w.r.t. classes.

Other methods use information present in the component schemas to relate attributes, entity types, and relationship types:

Navathe and Gadgil [41] compare object types by their names, the names of their key attributes, and the names of their non-key attributes.

Motro and Buneman [39, 38] compare classes by the names and the types of their attributes, using a given equivalence relation on the set of primitive classes.

Fankhauser, Kracker, and Neuhold [20] compare classes by the names and the types of their attributes and on the relevance ratios of their attributes.

De Souza [16] defines a tool that compares object types using their names, their identifiers, and their properties (attributes and relationships). The comparison yields three probabilities, which are combined into an overall probability using three weights. The weights can be adapted to cope with homonyms, synonyms, and conflicting properties. Furthermore, the comparison is extended with a comparison of the constraints and the neighbours (parents and children) in the generalisation hierarchy.

Rundensteiner [44] compares classes using a 'subtype' relation based on the names and the domains of their properties and on their populations.

Bouzeghoub and Comyn-Wattiau [10] compare attributes by name and type. Furthermore, entity types are compared by their names, their attributes, their constraints, and their populations. The comparison of entity types yields four similarity values. Inheritance rules and restructuring rules are used to enhance the similarity values of entity types.

Garcia-Solaco, Castellanos, and Saltor [22] compare schemas using structural and semantic information. The structural information is given by

the attribute names and the attribute types. The semantic information is obtained by a semantic enrichment phase, where populations of a schema are analysed to infer dependencies (e.g., keys) and the inferred dependencies are used to create a semantically rich schema with different kinds of aggregations and generalisations.

Larson, Navathe, and Elmasri [33] compare attributes by their domain, their cardinalities, the integrity constraints, the allowable operations, and their scale. Furthermore, object classes are compared by their key attributes.

Sheth, Larson, Cornelio, and Navathe [49, 50] define a tool that uses the attribute comparison of [33]. The tool derives an entity similarity matrix, where each element specifies the number of equivalent attributes between two entity classes. Using the information in this matrix, the tool asks the designer to specify assertions for every pair of similar entity classes. If the designer specifies conflicting assertions, he/she is asked to change the assertions.

Hayne and Ram [25] compare classes by their names, their identifiers, the names and the types of their attributes, their relationships, and the types of their transactions.

To rephrase, integration methods can be distinguished by the way they interact with the designer. Assertion-based methods first require the designer to compare every pair of elements in the component schemas and then derive similarities between elements. This approach is based on the argument that semantic equivalence can only be determined by the designer (with help of the end users). The advantage is that all similarities derived from the assertions reflect the real world semantics of the users; a disadvantage is that the designer has to do a lot of work. The other methods derive similarities between elements first and then require the designer to check these similarities. The advantage is that the amount of work for the designer is reduced; a disadvantage is that semantic equivalence is not determined in all cases.

1.2.2 Conforming and restructuring

We can distinguish between two types of integration methods based on the level of conflict resolution in the conforming phase and the level of redundancy removal in the restructuring phase. Some integration methods have a simple conforming phase and do the actual integration in the restructuring phase:

Biskup and Convent [9] and Casanova and Vidal [14] merge component schemas by superimposition, add the integration dependencies given by the designer, and normalise the resulting schema using transformations that remove redundant integration dependencies.

Motro and Buneman [39, 38], Dayal and Hwang [15], Larson, Navathe, and Elmasri [33], Buneman, Davidson, and Kosky [12], Rundensteiner [44], Sheth, Gala, and Navathe [46, 47] Sheth, Larson, Cornelio, and Navathe [49, 50], and Mannino, Navathe, and Effelsberg [36, 37] merge component schemas into a global specialisation hierarchy and normalise the hierarchy using algebraic operators, such as join and meet.

Other methods do the actual integration in the conforming phase and also have a restructuring phase:

Elmasri and Wiederhold [19], Elmasri and Navathe [18, 42, 40], Shoval and Zohn [52], and Spaccapietra, Parent, and Dupont [54, 56, 55] define several types of operators to integrate attributes and object types. Component schemas are conformed by applying the appropriate integration operators to the attributes and the object types.

Batini and Lenzerini [7] and Saltor, Castellanos, and Garcia-Solaco [45] define several types of operators that transform attributes and object types. These operators are used to transform the attributes and the object types of component schemas to enable merging.

To summarise, integration methods can be distinguished by the kind of transformations they use. Some methods only use transformations to remove redundancy in the restructuring phase, while other methods also use transformations to resolve conflicts in the conforming phase.

1.3 This thesis

The approach towards schema integration described in this thesis has been developed in the context of an object-oriented data model, which provides an integrated formalism for defining both structure and behaviour. The motivation for choosing an object-oriented data model is that we want to compare schemas not only by structure, but also by behaviour, since behaviour gives additional information about the schemas. Our approach [57, 58, 59, 60, 61] differs from other approaches on a number of points:

1. *Names.* The theoretical basis of our approach is invariant with respect to renaming of classes, attributes, and methods. This means that our approach is supplementary to an approach that uses names to find similarities, e.g., a lexicon-based approach.
2. *Structure.* We propose using schema transformations already in the comparison phase to detect transformational similarities between classes. This means that more (and more complex) similarities can be detected before interaction with the designer.

3. *Behaviour*. We also propose using semantic properties of methods to detect similarities between classes and syntactic properties of methods to reduce the computational complexity.

The main differences with other approaches are the combination of structure and behaviour and the application of transformations to detect similarities.

1.3.1 Outline

The outline of this thesis is as follows. Chapter 2 gives an example to illustrate the approach developed in Part I and Part II of this thesis. Part I introduces a simple data model and develops a theory for integrating simple database schemas based on morphisms between classes and semantic properties of methods. Part II extends the simple data model of Part I and adapts the theory developed in Part I to cope with the extended data model. Furthermore, Part II extends the theory with transformations on classes and syntactic properties of methods.

In Chapter 3, the first chapter of Part I, simple object-oriented database schemas are defined and formalised in terms of types and functions.

In Chapter 4, a number of equivalence relations for types are introduced and proven to be logically equivalent. The same is done for a number of subtyping relations.

In Chapter 5, an equality relation for functions is introduced and proven to be decidable. The same is done for a subfunction relation.

In Chapter 6, database schemas are compared by means of a subclass relation, which is defined in terms of the subtype and the subfunction relation.

In Chapter 7, database schemas are integrated by means of join operators w.r.t. the subclass relation.

In Chapter 8, the first chapter of Part II, the database schemas of Part I are extended with multiple inheritance for attributes, keys, query methods, method calls, and object creation.

In Chapter 9, several well-known type transformations are adapted to cope with recursive types. Furthermore, it is shown that these type transformations induce schema transformations.

In Chapter 10, a subset of the set of schema transformations is proven to be sound and complete w.r.t. data capacity.

In Chapter 11, schema integration is adapted to cope with extended database schemas. Furthermore, schema integration is extended with type transformations and syntactic properties of methods.

Finally, in Chapter 12, the overall approach is evaluated and directions for further research are given.

Chapter 2

An introductory example

In this chapter, we illustrate how structural transformations and behavioural properties can be used to integrate object-oriented database schemas. For that purpose, a windowing system and a desktop publishing system are introduced, which are inspired by the systems on page 44 and page 393 of Rumbaugh et al. [43]. A partial, graphical, definition of the systems is given by the schemas in Figure 2.1 and Figure 2.2.

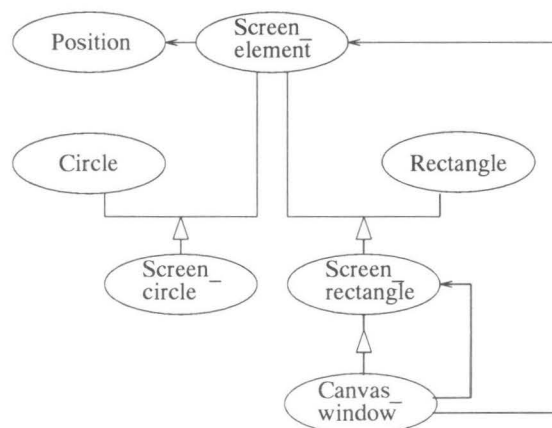


Figure 2.1: Partial schema for a windowing system.

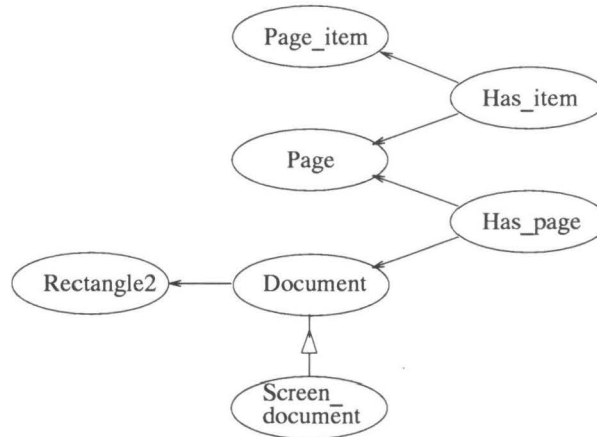


Figure 2.2: Partial schema for a desktop publishing system.

Object-oriented database schemas consist of classes, which are related by a subclass relation (cf. Galileo [2], Goblin [29], O₂ [34], and TM/FM [4]). In the figures, classes are represented by ellipses and subclass relationships by open arrows. For example, class *Screen_document* is a subclass of class *Document*.

A class consists of attributes, constraints, and methods, which are either inherited or new. The attributes specify the structure of the instances of the class and the methods specify the way the instances of the class are manipulated. An attribute has a name and a type, which can be a basic, set, or record type, or a class name *c*. In the latter case, the class to which the attribute belongs refers to the class with name *c*. This fact is represented by a simple arrow in the figures. For example, class *Document* refers to class *Rectangle2*.

A method has a name, a possibly empty list of parameters, an optional result, and a method body that consists of simple assignments. A method also has an implicit parameter (viz., an instance of the class to which the method belongs), which is referred to by **self** in the method body. There are a number of pre-defined functions associated with the basic types, such as addition of integers and concatenation of strings, and a number of pre-defined methods, such as **new**(*c*, *L*), which creates a new instance of the class named *c* according to list *L*.

Now, the windowing system and the desktop publishing system can be defined completely. The complete, textual, definition of the windowing system is given by the following schema:

Class Position

Attributes

x_co : integer

y_co : integer

Methods

```

create_position (x:integer,y:integer → p:Position) =
  p := new(Position, x_co=x, y_co=y)

```

Endclass

(* Instances of class Position model positions in a two-dimensional grid.

(* The attributes define the line numbers of a position.

(* The method creates a new position.

Class Screen_element**Attributes**

```

pos : Position

```

Methods

```

reset_pos (x:integer,y:integer) =
  self.pos := self.pos.create_position(x,y)

```

Endclass

(* Instances of Screen_element model objects that can be displayed on a

(* screen (grid of pixels). The method modifies the position of an object

(* by changing the reference to a position, not the position itself.

Class Rectangle**Attributes**

```

width : integer

```

```

height : integer

```

Constraints

```

width > 0

```

```

height > 0

```

Methods

```

area (→ a:integer) = a := self.width × self.height

```

Endclass**Class Screen_rectangle Isa Screen_element, Rectangle****Methods**

```

create_rectangle (x:integer,y:integer,w:integer,h:integer
  → s:Screen_rectangle) =
  var p : Position;
  p := p.create_position(x,y);
  s := new(Screen_rectangle, pos=p, width=w, height=h)
display () =
  rect(pos.x_co, pos.y_co, pos.x_co+width, pos.y_co+height)

```

Endclass

(* Instances of class Screen_rectangle model rectangles on a screen.

(* The class is a specialisation of both Screen_element and Rectangle.

(* The method displays a rectangle on the screen using special function

(* **rect**. Note that **self** has been omitted in the method body, e.g.,

(* **self.pos.x_co** has become **pos.x_co**.

Class Circle

Attributes

radius : integer

Constraints

radius > 0

Methods

area (\rightarrow a:rational) = a := $3.14 \times \text{radius} \times \text{radius}$

Endclass

Class Screen_circle **Isa** Screen_element, Circle

Methods

display () = **circ**(pos.x_co, pos.y_co, radius)

Endclass

(* Instances of class Screen_circle model circles on a screen.

(* The class is a specialisation of both Screen_element and Circle.

(* The method displays a circle on the screen using special function **circ**.)

Class Canvas_window **Isa** Screen_rectangle

Attributes

name : string

canvas : Screen_rectangle

items : {Screen_element}

Constraints

pos.x_co < canvas.pos.x_co

pos.y_co < canvas.pos.y_co

canvas.pos.x_co + canvas.width < pos.x_co + width

canvas.pos.y_co + canvas.height < pos.y_co + height

Methods

reset_canvas_pos (x:integer,y:integer) =

canvas := canvas.create_rectangle(x, y, canvas.width, canvas.height)

display () =

rect(pos.x_co, pos.y_co, pos.x_co+width, pos.y_co+height);

rect(canvas.pos.x_co, canvas.pos.y_co, canvas.pos.x_co+canvas.width,
canvas.pos.y_co+canvas.height)

move (x:integer,y:integer) =

var dx,dy : integer;

dx := canvas.pos.x_co - pos.x_co;

dy := canvas.pos.y_co - pos.y_co;

reset_pos(x,y);

reset_canvas_pos(x+dx,y+dy);

display()

Endclass

(* Instances of class Canvas_window model simple windows for drawing

(* pictures. The class is a specialisation of Screen_rectangle, because

(* instances of Screen_rectangle are windows in their simplest form.
 (* Attribute canvas models the part of the window that is meant for drawing
 (* pictures. The constraints model the fact that the position of the canvas
 (* is absolute and the fact that the pictures must be within the window.
 (* Method display is a specialisation of the method in class Screen_rectangle
 (* and displays the layout of a window.

The complete, textual, definition of the desktop publishing system is given by the following schema:

Class Page

Attributes

number : integer

Endclass

(* Instances of class Page model pages in a document.

Class Page.item

Attributes

pos : <x:integer,y:integer>

text : string

Endclass

(* Instances of class Page.item model items on a page.

Class Has.item

Attributes

page : Page

item : Page.item

Constraints

key item

Endclass

(* Instances of class Has.item model the fact that pages can have zero, one, or

(* more items. The constraint models the fact that every item can only belong

(* to one page.

Class Rectangle2

Attributes

length : integer

width : integer

Constraints

length > 0

width > 0

Endclass

Class Document

Attributes

name : string

author : string

```

    shape : Rectangle2
    margin : Rectangle2
Endclass
(* Instances of class Document model text documents.
(* Attribute margin defines the part of the page that is meant for writing text.
Class Has_page
Attributes
    document : Document
    page : Page
Constraints
    key page
Endclass
(* Instances of class Has_page model the fact that documents can have zero,
(* one, or more pages. The constraint models the fact that every page can
(* only belong to one document.
Class Screen_document Isa Document
Attributes
    pos : <x:integer,y:integer>
    margin_pos : <x:integer,y:integer>
Constraints
    pos.x < margin_pos.x
    pos.y < margin_pos.y
    margin_pos.x + margin.length < pos.x + shape.length
    margin_pos.y + margin.width < pos.y + shape.width
Methods
    reset_positions (x:integer,y:integer) =
        var dx,dy : integer;
        dx := margin_pos.x - pos.x;
        dy := margin_pos.y - pos.y;
        pos.x := x; pos.y := y;
        margin_pos.x := x+dx;
        margin_pos.y := y+dy
    display () =
        rectangle(pos.x, pos.y, pos.x+shape.length, pos.y+shape.width);
        rectangle(margin_pos.x, margin_pos.y, margin_pos.x+margin.length,
            margin_pos.y+margin.width)
    move (x:integer,y:integer) =
        reset_positions(x,y); display()
Endclass
(* Instances of class Screen_document model text documents that can be
(* displayed on a screen. The constraints model the fact that the position of
(* the margin is absolute and the fact that the text must be within the page.

```

(* Method display displays the layout of a document using special function
 (* rectangle.

2.1 Comparison

To compare classes, we transform classes in such a way that there is a subtype morphism or an optimal attribute mapping between them. Subtype morphisms are treated in Chapter 6, attribute mappings in Chapter 7, and transformations in Chapter 9. Transformations include aggregation of multiple attributes, objectification of a single attribute, and their inverse operations.

Using the inverse of objectification to replace a reference to a class by a record type and the inverse of aggregation to replace one attribute by a number of attributes, we can flatten Canvas_window and Screen_document as follows:

Class Canvas_window

Attributes

```
pos_x_co : integer
pos_y_co : integer
width : integer
height : integer
name : string
canvas_pos_x_co : integer
canvas_pos_y_co : integer
canvas_width : integer
canvas_height : integer
items : {Screen_element}
```

Constraints

```
width > 0
height > 0
canvas_width > 0
canvas_height > 0
pos_x_co < canvas_pos_x_co
pos_y_co < canvas_pos_y_co
canvas_pos_x_co + canvas_width < pos_x_co + width
canvas_pos_y_co + canvas_height < pos_y_co + height
```

Methods

```
reset_pos (x:integer,y:integer) =
  pos_x_co := x ; pos_y_co := y
area (→ a:integer) =
  a := width × height
create_rectangle (x:integer,y:integer,w:integer,h:integer
  → s:Screen_rectangle) =
```



```

s := new(Screen_rectangle, pos_x.co=x, pos_y.co=y,
        width=w, height=h)
reset_canvas_pos (x:integer,y:integer) =
  canvas_pos_x.co := x ; canvas_pos_y.co := y
display () =
  rect(pos_x.co, pos_y.co, pos_x.co+width, pos_y.co+height);
  rect(canvas_pos_x.co, canvas_pos_y.co, canvas_pos_x.co+canvas_width,
        canvas_pos_y.co+canvas_height)
move (x:integer,y:integer) =
  var dx,dy : integer;
  dx := canvas_pos_x.co - pos_x.co;
  dy := canvas_pos_y.co - pos_y.co;
  reset_pos(x,y);
  reset_canvas_pos(x+dx,y+dy);
  display()

```

Endclass**Class** Screen_document**Attributes**

```

name : string
author : string
shape_length : integer
shape_width : integer
margin_length : integer
margin_width : integer
pos_x : integer
pos_y : integer
margin_pos_x : integer
margin_pos_y : integer

```

Constraints

```

shape_length > 0
shape_width > 0
margin_length > 0
margin_width > 0
pos_x < margin_pos_x
pos_y < margin_pos_y
margin_pos_x + margin_length < pos_x + shape_length
margin_pos_y + margin_width < pos_y + shape_width

```

Methods

```

reset_positions (x:integer,y:integer) =
  var dx,dy : integer;
  dx := margin_pos_x - pos_x;
  dy := margin_pos_y - pos_y;

```

```

pos_x := x; pos_y := y;
margin_pos_x := x+dx;
margin_pos_y := y+dy
display () =
  rectangle(pos_x, pos_y, pos_x+shape_length, pos.y+shape_width);
  rectangle(margin_pos_x, margin_pos_y, margin_pos_x+margin_length,
    margin_pos_y+margin_width)
move (x:integer,y:integer) =
  reset_positions(x,y) ; display()

```

Endclass.

Attribute mappings between classes are partial functions between their attribute sets. An attribute mapping is optimal if it makes the sets of methods (and the sets of constraints) as much the same as possible. To find the optimal attribute mappings for class *Canvas_window* and class *Screen_document*, we make a list of the attribute sets that are used by the methods in *Canvas_window*:

1. reset_pos: pos_x_co, pos_y_co
2. area: width, height
3. create_rectangle: pos_x_co, pos_y_co, width, height
4. reset_canvas_pos: canvas_pos_x_co, canvas_pos_y_co
5. display: pos_x_co, pos_y_co, width, height, canvas_pos_x_co, canvas_pos_y_co, canvas_width, canvas_height
6. move: pos_x_co, pos_y_co, width, height, canvas_pos_x_co, canvas_pos_y_co, canvas_width, canvas_height

and a list of the attribute sets that are used by the methods in *Screen_document*:

1. reset_positions: pos_x, pos_y, margin_pos_x, margin_pos_y
2. display: shape_length, shape_width, margin_length, margin_width pos_x, pos_y, margin_pos_x, margin_pos_y
3. move: shape_length, shape_width, margin_length, margin_width pos_x, pos_y, margin_pos_x, margin_pos_y.

We use the cardinality of the attribute sets (and the types of the attributes in the attribute sets) to select pairs of methods that will be compared. A pair consists of a method in class *Canvas_window* and a method in class *Screen_document*, such that their attribute sets have the same cardinality:

1. create_rectangle and reset_positions

2. display and display
3. display and move
4. move and display
5. move and move.

In order to compare two methods w.r.t. a mapping, we need to know the effect of the pre-defined methods, the predefined functions, and the special functions in the application, e.g., given by a set of axioms.

Now, suppose that **rect**(x_1, y_1, x_2, y_2) and **rectangle**(x_1, y_1, x_2, y_2) have the same effect. Method `create_rectangle` and method `reset_positions` are not the same (whatever attribute mapping we use), because `create_rectangle` yields a result and `reset_positions` does not. Method `move` and method `display` are neither the same, because `move` modifies an object and `display` does not. However, the methods named `display` are the same if we use an attribute mappings that:

1. either maps `pos_x_co`, `pos_y_co`, `width`, `height`, `canvas_pos_x_co`, `canvas_pos_y_co`, `canvas_width`, `canvas_height` to `pos_x`, `pos_y`, `shape_length`, `shape_width`, `margin_pos_x`, `margin_pos_y`, `margin_length`, `margin_width`
2. or maps `pos_x_co`, `pos_y_co`, `width`, `height`, `canvas_pos_x_co`, `canvas_pos_y_co`, `canvas_width`, `canvas_height` to `margin_pos_x`, `margin_pos_y`, `margin_length`, `margin_width`, `pos_x`, `pos_y`, `shape_length`, `shape_width`.

The first of these mappings also makes the methods named `move` the same, whereas the second does not. Furthermore, the first mapping also makes the sets of constraints of class `Canvas_window` and class `Screen_document` the same, whereas the second results in a set of inconsistent constraints. Hence, the first attribute mapping is optimal.

In the same way, other classes can be compared. However, we are only interested in pairs of classes that are the same for a significant part. These pairs are:

1. `Canvas_window` and `Screen_document`
2. `Screen_rectangle` and `Screen_rectangle2`
3. `Rectangle` and `Rectangle2`.

2.2 Conforming

To conform classes, we use the optimal attribute mappings found in the comparison phase and the help of the designer. Using the optimal attribute mapping,

we can conform class `Canvas_window` and class `Screen_document` by renaming some of their attributes and methods. However, the resulting classes will be quite different from the original classes at the beginning of this chapter. It is more natural to choose either the original class `Canvas_window` or the original class `Screen_document` and conform the other class using the optimal attribute mapping. This is done by the designer.

Suppose the designer chooses `Canvas_window` and also decides to rename attribute `canvas` to `margin`. Then `Screen_document` is conformed as follows:

Class `Screen_document`

Attributes

```
name : string
author : string
pos : Position
width : integer
height : integer
margin : Screen_rectangle2
```

Constraints

```
width > 0
height > 0
pos.x_co < margin.pos.x_co
pos.y_co < margin.pos.y_co
margin.pos.x_co + margin.width < pos.x_co + width
margin.pos.y_co + margin.height < pos.y_co + height
```

Methods

```
reset_pos (x:integer,y:integer) =
  var dx,dy : integer
  dx := margin.pos.x_co - pos.x_co;
  dy := margin.pos.y_co - pos.y_co;
  pos := pos.create_position(x,y);
  margin := margin.create_rectangle(x+dx, y+dy, margin.width,
    margin.height)
display () =
  rect(pos.x_co, pos.y_co, pos.x_co+width, pos.y_co+height);
  rect(margin.pos.x_co, margin.pos.y_co, margin.pos.x_co+margin.width,
    margin.pos.y_co+margin.height)
move (x:integer,y:integer) =
  var dx,dy : integer;
  dx := margin.pos.x_co - pos.x_co;
  dy := margin.pos.y_co - pos.y_co;
  pos := pos.create_position(x,y);
  margin := margin.create_rectangle(x+dx, y+dy, margin.width,
    margin.height);
```

```

        display()
Endclass
Class Screen_rectangle2
Attributes
    pos : Position
    width : integer
    height : integer
Constraints
    width > 0
    height > 0
Methods
    create_rectangle (x:integer,y:integer,w:integer,h:integer
        → s:Screen_rectangle2) =
        var p : Position;
        p := p.create_position(x,y);
        s := new(Screen_rectangle, pos=p, width=w, height=h)
Endclass.

```

2.3 Merging and restructuring

To merge classes after they have been conformed, we use join operators to obtain the most specialised common superclasses and the help of the designer to give a name to these common superclasses. Join operators are treated in Chapter 7 and Chapter 11.

After the conforming step of the previous section, class `Canvas_window` and class `Screen_rectangle` can be merged as follows:

```

Class Window
Attributes
    name : string
    pos : Position
    width : integer
    height : integer
    margin : Screen_rectangle2
Constraints
    width > 0
    height > 0
    pos.x_co < margin.pos.x_co
    pos.y_co < margin.pos.y_co
    margin.pos.x_co + margin.width < pos.x_co + width
    margin.pos.y_co + margin.height < pos.y_co + height
Methods

```

```

reset_pos (x:integer,y:integer) =
  pos := pos.create_position(x,y)
display () =
  rect(pos.x_co, pos.y_co, pos.x_co+width, pos.y_co+height);
  rect(margin.pos.x_co, margin.pos.y_co, margin.pos.x_co+margin.width,
    margin.pos.y_co+margin.height)
move (x:integer,y:integer) =
  var dx,dy : integer;
  dx := margin.pos.x_co - pos.x_co;
  dy := margin.pos.y_co - pos.y_co;
  pos := pos.create_position(x,y);
  margin := margin.create_rectangle(x+dx, y+dy, margin.width,
    margin.height);
  display()
Endclass
Class Canvas_window Isa Window
Attributes
  margin : Screen_rectangle
  items : {Screen_element}
Methods
  area (→ a:integer) = a := width × height
  create_rectangle (x:integer,y:integer,w:integer,h:integer
    → s:Screen_rectangle) =
    var p : Position;
    p := p.create_position(x,y);
    s := new(Screen_rectangle, pos=p, width=w, height=h)
  reset_margin_pos (x:integer,y:integer) =
    margin := margin.create_rectangle(x, y, margin.width, canvas.height)
Endclass
Class Screen_document Isa Window
Attributes
  author : string
Methods
  reset_pos (x:integer,y:integer) =
    var dx,dy : integer;
    dx := margin.pos.x_co - pos.x_co;
    dy := margin.pos.y_co - pos.y_co;
    pos := pos.create_position(x,y);
    margin := margin.create_rectangle(x+dx, y+dy, margin.width,
      margin.height)
Endclass.

```

In the same way, the other pairs of classes found in the comparison phase can

be merged. The merged schema can be restructured to reduce redundancy and to improve understandability. Again, interaction with the designer is needed, because the designer has to decide about understandability. For example, method area can be migrated from class `Canvas_window` to class `Window` or class `Window` and class `Document` can be factorised into class `Frame`:

Class Frame

Attributes

name : string
width : integer
height : integer
margin : Rectangle2

Constraints

width > 0
height > 0

Endclass.

We conclude this chapter by giving a partial, graphical, representation of the integrated schema in Figure 2.3.

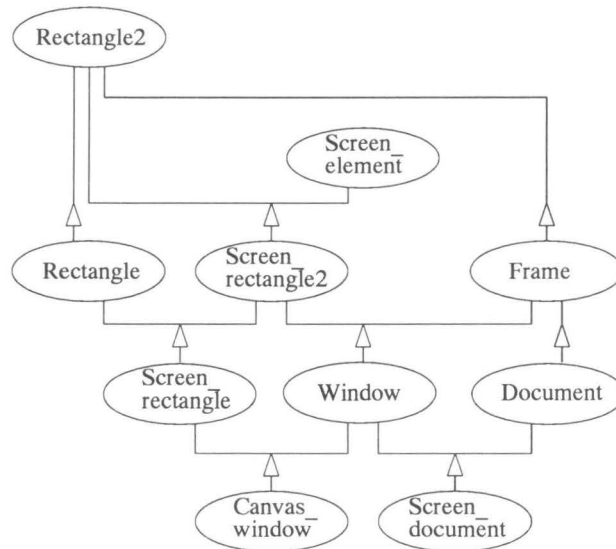


Figure 2.3: Partial schema for an integrated windowing and desktop publishing system.

The integrated schema is correct in the sense that any instance of the original schemas can be translated into an instance of the integrated schema without loss of information.

Part I

Chapter 3

Database schemas

In this part, we introduce an approach towards schema integration based on morphisms between classes and semantic properties of methods [57, 58]. In the following part, we will extend the approach using transformations on classes and syntactic properties of methods. The approach is developed in the context of an object-oriented data model, similar to Galileo [2], Goblin [29, 30], GOOD [23], O₂ [34], and TM/FM [4, 5]. Such a data model offers an integrated formalism for defining both structure and behaviour of objects in the real world. The motivation for choosing an object-oriented data model is that we want to compare schemas not only by structure, but also by behaviour, because we think that behaviour is a very important aspect of ‘real world objects’.

A schema in an object-oriented data model is a class hierarchy: a set of classes, which are related by a subclass relation and consist of attributes, constraints and methods. In this part, we consider classes with attributes and update methods only. A class in such a schema corresponds to a ‘real world class’, the states of which are sets of real world objects. The state of a real world class is represented by a set of class instances, which correspond to real world objects. The attributes of a class instance correspond to ‘structural’ properties of a real world object and the update methods of a class instance correspond to ‘behavioural’ properties of a real world object. The distinction between attributes and update methods is useful, because the attributes of a class instance, which can vary from instance to instance, define the state of a real world object, whereas the update methods of a class instance, which are the same for all instances, define the possible transitions from the state of a real world object.

There are several options to formalise class hierarchies and class extensions. One option is to assign a set of simple instances (e.g., names) to a class and to formalise every attribute as a variable function from the set of instances to the corresponding codomain, and every update method as a fixed function from the

set of instances and the domains of the parameters to the set of instances (cf. DAPLEX [51]). An instance can be queried and manipulated by applying the formalised attributes and methods.

We have chosen another option, similar to the approach of TM/FM [5], where the attributes of a class and a special identifier attribute are aggregated into a record type (the underlying type of the class). The set of possible instances of a class is the set of instances of its underlying type. Furthermore, every update method is formalised as a lambda expression, the interpretation of which is a function from the set of instances and the domains of the parameters to the set of instances. Note that the distinction between types and functions follows the useful distinction between attributes and update methods.

Our formalisation differs from the approach of [5] as follows. In [5], identifier attributes are used both to discriminate between different instances and to cope with recursive class definitions. In our formalisation, identifier attributes are used to discriminate between different instances and recursive types are used to cope with recursive class definitions. In fact, our formalisation resembles the approach of GOOD [23], where cyclic graphs are used to model database schemas and database instances, except that we use special identifier attributes. The resemblance stems from the fact that types and instances of types can be interpreted as cyclic graphs or infinite trees. In the remainder of this chapter, we define simple class hierarchies and formalise them in terms of underlying types and functional forms.

3.1 Syntax

In this section, we introduce simple database schemas, consisting of classes with attributes and update methods. An attribute has a name and a type, which can be a basic, set, or record type, or a class name. An update method has a name, a possibly empty list of parameters, and a body, which consists of simple assignments.

Definition 3.1 First, five disjoint sets are postulated: a set CN of class names, a set AN of attribute names, a set MN of method names, a set L of labels, and a set BC of basic constants (i.e., ‘integer’, ‘rational’, and ‘string’ constants). The sets are generated by the nonterminals CN , AN , MN , L , and BC , respectively. Simple class hierarchies are the sentences of the following BNF-grammar, where the plus sign (+) denotes a finite, nonempty, repetition, square brackets ([]) denote an option, and the vertical bar (|) denotes a choice:

```

Hierarchy ::= Class+
Class      ::= ‘Class’ CN [ ‘Isa’ CN+ ]
              [ ‘Attributes’ Att+ ]
              [ ‘Methods’ Meth+ ]

```

		'Endclass'
Att	::=	AN ':' Type
Type	::=	BasicType SetType RecordType CN
BasicType	::=	'integer' 'rational' 'string'
SetType	::=	'{' Type '}'
RecordType	::=	'<' FieldList '>'
FieldList	::=	Field Field ',' FieldList
Field	::=	L ':' Type
Meth	::=	MN ['(' ParList ')'] '=' AsnList
ParList	::=	Par Par ',' ParList
Par	::=	L ':' BasicType
AsnList	::=	Assign Assign ';' AsnList
Assign	::=	Dest ':=' Source 'insert(' Source ',' Dest ')
Dest	::=	AN Dest ':' L
Source	::=	Term Term '+' Source Term '-' Source Term '×' Source Term '÷' Source
Term	::=	BC L 'self' Path
Path	::=	AN Path '.' L Path '.' AN

An assignment of the form **'insert(e, V)'** is to be interpreted as $V := V \cup \{e\}$.
□

A class hierarchy is well-defined if it satisfies three conditions. The first condition is that class names are unique within the hierarchy, and classes only refer to classes in the hierarchy, and the **Isa** relation is acyclic. The second condition is that attributes have a unique name within their class and are well-typed (see section on underlying types). The third is that methods have a unique name within their class and are well-typed (see section on functional forms).

3.2 Underlying types

In this section, we consider the attribute part of a class definition. We define underlying types of classes and attribute specialisation.

Example 3.2 The following class hierarchy introduces a class **'SimpleAddress'** and a class **'Address'**, which inherits from class **'SimpleAddress'** and adds a new attribute.

```

Class SimpleAddress
Attributes
  house : integer
  street : string
  city : string

```

```

Endclass
Class Address Isa SimpleAddress
Attributes
  country : string
Endclass.

```

□

In definitions, we abbreviate every class to a 4-tuple $C = (c, S, A, M)$, where c is the name of the class, S is the set of names of its superclasses, A is the set of its new attributes, and M is the set of its new methods. The name of (abbreviated) class C is denoted by $name(C)$ and the set of names of its superclasses is denoted by $sup_names(C)$. Furthermore, we abbreviate a class hierarchy to the set of abbreviations of the classes in the class hierarchy.

The set of all attributes of a class consists of both the new and inherited attributes. In order to formalise this, we formalise the **Isa** relation between classes first.

Definition 3.3 Let H be an abbreviated class hierarchy satisfying the first condition for well-defined class hierarchies. The subclass relation on H , denoted by $sub(H)$, is defined as the reflexive and transitive closure of:

$$isa(H) = \{(name(C_1), name(C_2)) \mid C_1 \in H \wedge C_2 \in H \wedge name(C_2) \in sup_names(C_1)\}.$$

Relation $isa(H)$ is acyclic if its transitive closure only contains pairs of the form (c_1, c_2) , such that $c_1 \neq c_2$. □

Now we can define the set of attributes of a class.

Definition 3.4 Let H be a class hierarchy satisfying the first condition and $C = (c, S, A, M)$ be a class in H . The set of all attributes of C , denoted by $atts(C)$, is defined as:

$$atts(C) = A \cup \{a : T \mid \exists C' \in H [(name(C), name(C')) \in isa(H) \wedge a : T \in atts(C')] \wedge \forall a' : T' \in A[a \neq a']\}.$$

□

Note that redefined attributes override the corresponding attributes in the superclass. Furthermore, note that a class can inherit attributes from more than one superclass (multiple inheritance). However, the second condition for well-defined class hierarchies requires that inherited attributes with the same name must be the same (or must be redefined).

Every class in a class hierarchy has an underlying type, which describes the structure of the class, i.e., the structure of the objects in its extensions (cf.

TM/FM [4, 5]). Informally, the underlying type of a class is an aggregation of its attributes, where recursive types [3] are used to cope with attributes that refer to classes.

Definition 3.5 First, we postulate a new type ‘oid’, the extension of which is an enumerable set of object identifiers.

Let H be a class hierarchy satisfying the first condition, C be a class in H , and c be the name of C . The underlying type of class C , denoted by $type(C)$, is defined as:

$$type(C) = \tau(c, \emptyset)$$

where

$$\begin{aligned} \tau(d, \eta) &= \mu d . < id : oid, a_1 : \tau(T_1, \eta \cup \{d\}), \dots, a_k : \tau(T_k, \eta \cup \{d\}) > \\ &\quad \text{if } d \notin \eta \text{ and } \exists D \in H[name(D) = d \wedge atts(D) = \{a_1 : T_1, \dots, a_k : T_k\}], \\ \tau(d, \eta) &= d \text{ if } d \in \eta, \\ \tau(B, \eta) &= B \text{ if } B \in \{\text{integer, rational, string}\}, \\ \tau(\{U\}, \eta) &= \{\tau(U, \eta)\}, \\ \tau(< l_1 : U_1, \dots, l_n : U_n >, \eta) &= < l_1 : \tau(U_1, \eta), \dots, l_n : \tau(U_n, \eta) >, \end{aligned}$$

where μ is the minimal type constructor and d has become a type variable. Type $\mu t. \alpha$ should be interpreted as the ‘smallest’ type τ , such that τ is equivalent to $\alpha[t \setminus \tau]$. Set η contains the names of the classes for which a (recursive) type is being constructed as part of the construction of the underlying type of class C . If η contains d , then $\tau(d, \eta) = d$ starts the recursion. \square

Note that the underlying type of a class depends on the hierarchy.

Example 3.6 The underlying types of class ‘SimpleAddress’ and class ‘Address’ of Example 3.2 are given by:

$$\begin{aligned} \tau_S &= \mu t_S . < id : oid, house : integer, street : string, city : string > \\ \tau_A &= \mu t_A . < id : oid, house : integer, street : string, city : string, country : string >, \end{aligned}$$

where t_S denotes ‘SimpleAddress’ and t_A denotes ‘Address’. We have obtained underlying types that are the same as the underlying types in TM/FM. \square

To reformulate the second condition for well-defined class hierarchies, we have to introduce well-typed attributes and a specialisation relation on attributes. An attribute is well-typed if its type is well-defined.

Definition 3.7 Let H be a class hierarchy, C be a class in H , and $a : T$ be an attribute in $atts(C)$. Then $a : T$ is well-typed if and only if $T \in WTypes$, where $WTypes$ is the smallest set, such that:

1. if $c \in CN$, then $c \in WTypes$

2. if $B \in \{\text{integer}, \text{rational}, \text{string}\}$, then $B \in WTypes$
3. if $U \in WTypes$, then $\{U\} \in WTypes$
4. if $\{U_1, \dots, U_n\} \subseteq WTypes$ and $\{l_1, \dots, l_n\}$ is a set of n distinct labels in L , then $\langle l_1 : U_1, \dots, l_n : U_n \rangle \in WTypes$.

□

An attribute is a specialisation of another attribute if the type of the first is an extension of the type of the second.

Definition 3.8 Let H be a class hierarchy, C_1 and C_2 be classes in H , $a : T_1$ be an attribute in $atts(C_1)$, and $a : T_2$ be an attribute in $atts(C_2)$. Then $a : T_1$ is a specialisation of $a : T_2$ if and only if:

1. $name(C_1) \leq_H name(C_2)$
2. $T_1 \leq_H T_2$

where \leq_H is defined by:

$$\begin{aligned}
 d &\leq_H d' \text{ if } (d, d') \in sub(H) \\
 B &\leq_H B \text{ if } B \in \{\text{integer}, \text{rational}, \text{string}\}, \\
 \{U\} &\leq_H \{U'\} \text{ if } U \leq_H U', \\
 \langle l_j : U_j \mid j \in J \rangle &\leq_H \langle l_j : U'_j \mid j \in J' \rangle \text{ if } J \supseteq J' \wedge \forall j \in J' [U_j \leq_H U'_j].
 \end{aligned}$$

□

The specialisation relation on attributes is a simple subtype relation. In Chapter 4, more complex subtype relations will be defined.

Attribute specialisation is a kind of attribute redefinition, where an inherited attribute $a : T$, defined in class C , is replaced by $a : T'$ in subclass C' , such that $a : T'$ is a specialisation of $a : T$. If attribute specialisation is allowed, it still holds that the underlying type of a subclass is a subtype of the underlying type of the superclass (in case of general redefinition of attributes, it does not hold).

Finally, we can reformulate the second condition for well-defined class hierarchies: every attribute must have a unique name within its class, the type of every attribute must be well-defined, and every inherited attribute must be a specialisation of the corresponding attribute in the superclass.

Example 3.9 The following class hierarchy introduces a class ‘Person’ and a class ‘Employee’, which specialises inherited attribute ‘address’ and adds attribute ‘company’.

Class Person
Attributes
 name : string

```

    mother : Person
    address : SimpleAddress
Endclass
Class Employee Isa Person
Attributes
    address : Address
    company : string
Endclass.

```

The underlying types of class ‘Person’ and class ‘Employee’ are given by:

$$\tau_P = \mu t_P . \langle \text{id:oid}, \text{name:string}, \text{mother:}t_P, \text{address:}\tau_S \rangle$$

$$\tau_E = \mu t_E . \langle \text{id:oid}, \text{name:string}, \text{mother:}\tau_P, \text{address:}\tau_A, \text{company:string} \rangle,$$

where t_P denotes ‘Person’, t_E denotes ‘Employee’, and τ_S and τ_A are the underlying types of class ‘Simple_Address’ and class ‘Address’. The underlying types of class ‘Person’ and class ‘Employee’ are not the same as the underlying types in TM/FM, which are $\langle \text{id:oid}, \text{name:string}, \text{mother:oid}, \text{address:oid} \rangle$ and $\langle \text{id:oid}, \text{name:string}, \text{mother:oid}, \text{address:oid}, \text{company:string} \rangle$, respectively. \square

3.3 Functional forms

In this subsection, we consider the method part of a class definition. We define functional forms of methods and method specialisation.

Example 3.10 The following class hierarchy introduces a class ‘Person1’ with a method ‘change_addr’ and a class ‘Person2’ with a method ‘move’.

```

Class Person1
Attributes
    name : string
    addr : <house:integer,street:string>
Methods
    change_addr (h:integer,s:string) =
        addr.house := h;
        addr.street := s
Endclass
Class Person2
Attributes
    name : string
    address : <house:integer,street:string,city:string>
Methods
    move (h:integer,s:string,c:string) =
        address.house := h;

```



```

        address.street := s;
        address.city := c
    Endclass.

```

□

The set of all methods of a class in a class hierarchy consists of both the new and inherited methods.

Definition 3.11 Let H be a class hierarchy satisfying the first condition and $C = (c, S, A, M)$ be a class in H . The set of all methods of C , denoted by $meths(C)$, is defined as:

$$meths(C) = M \cup \{m(P) = E \mid \exists C' \in H [(name(C), name(C')) \in isa(H) \wedge m(P) = E \in meths(C')] \wedge \forall m'(P') = E' \in M [m \neq m']\}.$$

□

Note that redefined methods override the corresponding methods in the super-class. Furthermore, note that a class can inherit methods from more than one superclass (multiple inheritance). However, the third condition for well-defined class hierarchies requires that inherited methods with the same name must be the same (or must be redefined).

Every class in a class hierarchy has a set of functional forms (one for each of its methods), which describe the way in which the objects in the extensions of the class are manipulated (cf. TM/FM [4, 17]). Informally, the functional form of a method is a function whose body is an accumulation of the assignments in the method body.

Definition 3.12 Let H be a class hierarchy satisfying the first and second condition and C be a class in H . Furthermore, let $\{a_1 : T_1, \dots, a_k : T_k\}$ be $atts(C)$ and $m(P) = E$ be an element of $meths(C)$. The functional form of method $m(P) = E$ in class C , denoted by $func(C, m(P) = E)$, is defined as:

$$func(C, m(P) = E) = \lambda obj : type(C) \lambda P . eval(E)(\mu \delta(obj). < id=obj.id, a_1=\sigma_0(a_1, T_1), \dots, a_k=\sigma_0(a_k, T_k) >)$$

where

$$\begin{aligned} \sigma_0(r, d) &= obj.r \text{ if } d \in CN \\ \sigma_0(r, B) &= obj.r \text{ if } B \in \{\text{integer, rational, string}\} \\ \sigma_0(r, \{U\}) &= obj.r \\ \sigma_0(r, < l_1 : U_1, \dots, l_n : U_n >) &= < l_1 = \sigma_0(r.l_1, U_1), \dots, l_n = \sigma_0(r.l_n, U_n) > \end{aligned}$$

and

$$\begin{aligned}
eval(L_1; L_2)\sigma &= eval(L_2)(eval(L_1)\sigma) \\
eval(a_1 := s)(obj) &= \\
&\quad \mu \delta(obj). \langle id = e_0, a_1 = ev(s, obj), a_2 = e_2, \dots, a_k = e_k \rangle \\
eval(a_1.l_1 \dots l_n := s)(obj) &= \\
&\quad \mu \delta(obj). \langle id = e_0, a_1 = e.a_1[l_1 \dots l_n = ev(s, obj)], a_2 = e_2, \dots, a_k = e_k \rangle \\
eval(insert(s, a_1))(obj) &= \\
&\quad \mu \delta(obj). \langle id = e_0, a_1 = e.a_1 \cup \{ev(s, obj)\}, a_2 = e_2, \dots, a_k = e_k \rangle \\
eval(insert(s, a_1.l_1 \dots l_n))(obj) &= \\
&\quad \mu \delta(obj). \langle id = e_0, a_1 = e.a_1[l_1 \dots l_n = e.a_1.l_1 \dots l_n \cup \{ev(s, obj)\}], \\
&\quad \quad a_2 = e_2, \dots, a_k = e_k \rangle
\end{aligned}$$

and

$$\begin{aligned}
ev(b, obj) &= b \\
&\quad \text{if } b \in BC \\
ev(l, obj) &= l \\
&\quad \text{if } l \in L, \\
ev(\mathbf{self}, obj) &= \delta(obj), \\
ev(l_1 \dots l_n, obj) &= obj.l_1 \dots l_n \\
&\quad \text{if } l_1 \in AN, \\
ev(s_1 \theta s_2, obj) &= ev(s_1, obj) \theta ev(s_2, obj) \\
&\quad \text{if } \theta \in \{+, -, \times, \div\}.
\end{aligned}$$

□

Example 3.13 Let C_{P1} be class ‘Person1’ and C_{P2} be class ‘Person2’ of Example 3.10. Let $meth_1$ be method ‘change_addr’ in class ‘Person1’ and $meth_2$ be method ‘move’ in class ‘Person2’. The functional forms of method ‘change_addr’ in class ‘Person1’ and method ‘move’ in class ‘Person2’ are given by:

$$\begin{aligned}
func(C_{P1}, meth_1) &= \lambda o:\tau_{P1} \lambda h:\text{integer} \lambda s:\text{string} . \\
&\quad \mu \delta(o). \langle id=o.id, name=o.name, addr=\langle house=h, street=s \rangle \rangle, \\
func(C_{P2}, meth_2) &= \lambda o:\tau_{P2} \lambda h:\text{integer} \lambda s:\text{string} \lambda c:\text{string} . \\
&\quad \mu \delta(o). \langle id=o.id, name=o.name, address=\langle house=h, street=s, city=c \rangle, \\
&\quad \quad company=o.company, salary=o.salary \rangle,
\end{aligned}$$

where τ_{P1} is the underlying type of class ‘Person1’ and τ_{P2} is the underlying type of class ‘Person2’. □

To reformulate the third condition, we have to introduce well-typed methods and a specialisation relation on methods. A method is well-typed if every assignment in its body is well-defined.

Definition 3.14 Let H be a class hierarchy, C be a class in H , and $m(P) = E$ be a method in $meths(C)$. An assignment $d := s$ (resp., $\mathbf{insert}(s, d)$) in E is

well-defined if the type of s is a subtype of the type of d according to \leq_H (resp., if the type of d is $\{T\}$ and the type of s is a subtype of T), where the type of a source or a destination is defined as follows:

1. the type of an integer constant is integer
2. the type of a rational constant is rational
3. the type of a string constant is string
4. the type of l is B if $l : B$ is a parameter in P
5. the type of **self** is $name(C)$
6. the type of $a.r_1 \dots r_n$ is $T.r_1 \dots r_n$ if $a : T$ is an attribute in $atts(C)$ and $T.r_1 \dots r_n \neq \perp$
7. the type of a complex term follows from the types of the subterms and the standard rules for $+$ (addition for integers and rationals; concatenation for strings), $-$ (subtraction for integers and rationals), \times (multiplication for integers and rationals), and \div (division for rationals only)
8. otherwise, the type of a source or a destination is undefined

and

$$\begin{aligned}
 & \langle l_1 : T_1, \dots, l_n : T_n \rangle . l = T \\
 & \quad \text{if } l \in \{l_1, \dots, l_n\} \wedge T = T_i \\
 & \langle l_1 : T_1, \dots, l_n : T_n \rangle . l = \perp \\
 & \quad \text{if } l \notin \{l_1, \dots, l_n\} \\
 & c'.a = T \\
 & \quad \text{if } \exists C' \in H \exists a' : T' \in atts(C') [c' = name(C') \wedge a = a' \wedge T = T'] \\
 & c'.a = \perp \\
 & \quad \text{if } \forall C' \in H \forall a' : T' \in atts(C') [c' \neq name(C') \vee a \neq a'].
 \end{aligned}$$

Finally, $m(P) = E$ is well-typed if and only if every assignment in E is well-defined. \square

A method is a specialisation of another method if the type of the first is an extension of the type of the second and the body of the first is an extension of the body of the second.

Definition 3.15 Let H be a class hierarchy, C_1 and C_2 be classes in H , $m(P_1) = E_1$ be a method in $meths(C_1)$, and $m(P_2) = E_2$ be a method in $meth(C_2)$. Then $m(P_1) = E_1$ is a specialisation of $m(P_2) = E_2$ if and only if:

1. $name(C_1) \leq_H name(C_2)$
2. $\tilde{E}_1 = E_2$

where \tilde{E}_1 is obtained from E_1 by removing any assignment $d := s$, such that d is not well-typed in C_2 . \square

The specialisation relation on methods is a simple subfunction relation. In Chapter 5, more complex subfunction relations will be defined.

Method specialisation is a kind of method redefinition, where an inherited method $m(P) = E$, defined in class C , is replaced by $m(P') = E'$ in subclass C' , such that $m(P') = E'$ is a specialisation of $m(P) = E$.

Finally, we can reformulate the third condition: every method must have a unique name within its class, every method must be well-typed, and every inherited method must be a specialisation of the corresponding method in the superclass.

Example 3.16 The following class hierarchy introduces a class ‘Person’ and a class ‘Employee’, which specialises inherited attribute ‘addr’ and inherited method ‘move’.

```

Class Person
Attributes
  name : string
  addr : <house:integer,street:string>
Methods
  move (h:integer,s:string) =
    addr.house := h;
    addr.street := s
Endclass
Class Employee Isa Person
Attributes
  addr : <house:integer,street:string,city:string>
Methods
  move (h:integer,s:string,c:string) =
    addr.house := h;
    addr.street := s;
    addr.city := c
Endclass.

```

\square

Chapter 4

Properties of underlying types

In the previous chapter, we used types to formalise states of real world objects and functions to formalise transitions between states. Since we think it is useful to distinguish between states and transitions, we treat types and functions separately. In this chapter, we focus on types and instances of types to obtain a formal basis for the comparison of attributes. In the following chapter, we will focus on functions to obtain a formal basis for the comparison of methods.

Traditional type systems consider functional types, which leads to complex semantics for the system (cf. the systems in the λ -cube [6] and the system of recursive types in [3]). In this thesis, we consider basic types, set types, record types, and recursive record types, which have simple set-theoretic semantics. We introduce structural type equivalence, similar to type equivalence in Algol68 ([32]), where types are represented by infinite trees. The motivation for trees is that they give a natural interpretation for recursive types. Furthermore, we define extensional type equivalence using extensions, derivable type equivalence using a derivation system, and reducible type equivalence based on a normalisation process. The motivation for extensions (i.e., sets of instances) is that they give a natural set-theoretic interpretation for types. The derivation system induces an algorithm that is similar to the algorithm in [32]. In fact, our approach resembles the approach of Amadio and Cardelli [3]: structural type equivalence resembles $=_T$, extensional type equivalence resembles $=_M$, reducible type equivalence resembles $=_R$, and derivable type equivalence $=_A$. In addition, we introduce derivable subtyping using a derivation system, structural subtyping using trees, and extensional subtyping using extensions. These subtype relations resemble subtype relations $<_A$, $<_T$, and $<_M$ from [3], respectively. However, there is an important difference. Since we have no functional types, but only basic, set, record, and re-

cursive record types, the models in our approach (viz., the extensions) are simpler. As a consequence, we have stronger results: derivable equivalence is sound and complete w.r.t. both structural and extensional type equivalence, and derivable subtyping is sound and complete w.r.t. both structural and extensional subtyping. Finally, we show that the equivalence relations and the subtype relations are polynomial time decidable [27]. These results are important for the comparison of attributes.

4.1 Syntax and semantics

In this section, we define types and instances of types syntactically and semantically. The set of types consists of type variables, basic types, set types, record types, and recursive record types. The set of μ -complete types is the same as the set of types, except that every record type is preceded by an occurrence of minimisation operator μ [3]. In fact, μ -completeness is only a technical restriction, since every general type can be rewritten as a μ -complete type. The restriction to μ -complete types just simplifies the proofs of a number of theorems. The syntax of types is given by the following definition.

Definition 4.1 First, two disjoint sets are postulated: a set *TypeVar* of type variables and a set \mathcal{L} of labels. The set of basic types, denoted by *BTypes*, is defined as {oid, integer, rational, string}. The set of types, denoted by *Types*, is inductively defined by:

1. if $t \in \textit{TypeVar}$, then $t \in \textit{Types}$
2. if $B \in \textit{BTypes}$, then $B \in \textit{Types}$
3. if $v \in \textit{Types}$, then $\{v\} \in \textit{Types}$
4. if $\{l_1, \dots, l_n\} \subseteq \mathcal{L}$ is a set of n distinct labels and $\{v_1, \dots, v_n\} \subseteq \textit{Types}$, then $\langle l_1 : v_1, \dots, l_n : v_n \rangle \in \textit{Types}$
5. if $t \in \textit{TypeVar}$ and $\{l_1, \dots, l_n\} \subseteq \mathcal{L}$ is a set of n distinct labels and $V = \{v_1, \dots, v_n\} \subseteq \textit{Types}$ and $\forall \tau \in V[t \notin \textit{bvars}(\tau)]$, then $\mu t. \langle l_1 : v_1, \dots, l_n : v_n \rangle \in \textit{Types}$,

where $\textit{bvars}(\tau)$ is the set of bound type variables in τ :

$$\begin{aligned}
 \textit{bvars}(t) &= \emptyset \text{ if } t \in \textit{TypeVar} \\
 \textit{bvars}(B) &= \emptyset \text{ if } B \in \textit{BTypes} \\
 \textit{bvars}(\{v\}) &= \textit{bvars}(v) \\
 \textit{bvars}(\langle l_1 : v_1, \dots, l_n : v_n \rangle) &= \textit{bvars}(v_1) \cup \dots \cup \textit{bvars}(v_n) \\
 \textit{bvars}(\mu t. \alpha) &= \textit{bvars}(\alpha) \cup \{t\}.
 \end{aligned}$$

Furthermore, the set of closed types, denoted by $CTypes$, is defined as follows:

$$CTypes = \{\tau \in Types \mid fvars(\tau) = \emptyset\},$$

where $fvars(\tau)$ is the set of free type variables in τ :

$$\begin{aligned} fvars(t) &= \{t\} \text{ if } t \in TypeVar \\ fvars(B) &= \emptyset \text{ if } B \in BTypes \\ fvars(\{v\}) &= fvars(v) \\ fvars(\langle l_1 : v_1, \dots, l_n : v_n \rangle) &= fvars(v_1) \cup \dots \cup fvars(v_n) \\ fvars(\mu t. \alpha) &= fvars(\alpha) - \{t\}. \end{aligned}$$

The set of μ -complete types, denoted by $\mu Types$, is inductively defined by:

1. if $t \in TypeVar$, then $t \in \mu Types$
2. if $B \in BTypes$, then $B \in \mu Types$
3. if $v \in \mu Types$, then $\{v\} \in \mu Types$
4. if $t \in TypeVar$ and $\{l_1, \dots, l_n\} \subseteq \mathcal{L}$ is a set of n distinct labels and $V = \{v_1, \dots, v_n\} \subseteq \mu Types$ and $\forall \tau \in V [t \notin bvars(\tau)]$, then $\mu t. \langle l_1 : v_1, \dots, l_n : v_n \rangle \in \mu Types$.

Finally, the set of closed μ -complete types, denoted by $C\mu Types$, consists of the μ -complete types that have no free type variables:

$$C\mu Types = \{\tau \in \mu Types \mid fvars(\tau) = \emptyset\}.$$

□

Strict type equality is defined as syntactic equality, modulo permutation of fields in record types. Weak type equality is defined as strict type equality, modulo addition of dummy type variables.

Definition 4.2 Let τ_1 and τ_2 be types. Strict equality of τ_1 and τ_2 , denoted by $\tau_1 = \tau_2$, is inductively defined as follows:

1. $t = t$ if $t \in TypeVar$
2. $B = B$ if $B \in BTypes$
3. $\{v\} = \{v'\}$ if $v = v'$
4. $\langle l_1 : v_1, \dots, l_n : v_n \rangle = \langle l'_1 : v'_1, \dots, l'_n : v'_n \rangle$ if $\forall i \in \{1, \dots, n\} \exists j \in \{1, \dots, n\} [l_i = l'_j \wedge v_i = v'_j]$
5. $\mu t. \alpha = \mu t'. \alpha'$ if $t = t'$ and $\alpha = \alpha'$.

Weak equality of τ_1 and τ_2 , denoted by $\tau_1 \approx \tau_2$, is inductively defined as follows:

1. $t \approx t$ if $t \in TypeVar$
2. $B \approx B$ if $B \in BTypes$
3. $\{v\} \approx \{v'\}$ if $v \approx v'$
4. $\langle l_1 : v_1, \dots, l_n : v_n \rangle \approx \langle l'_1 : v'_1, \dots, l'_n : v'_n \rangle$
if $\forall i \in \{1, \dots, n\} \exists j \in \{1, \dots, n\} [l_i = l'_j \wedge v_i \approx v'_j]$
5. $\mu t. \alpha \approx \mu t'. \alpha'$ if $t \approx t'$ and $\alpha \approx \alpha'$
6. $\alpha \approx \mu t. \alpha$ if $t \notin fvars(\alpha)$
7. $\mu t. \alpha \approx \alpha$ if $t \notin fvars(\alpha)$.

□

Using rule 6 and 7 of weak type equality, we can rewrite every type as a μ -complete type.

The semantics of types can be defined in terms of trees or extensions. The tree of a type represents the structure of the type.

Definition 4.3 Let τ be a type. The tree representing τ is defined as $struc(\tau, \emptyset)$, where $struc(\tau, \Gamma)$ is defined as follows:

$$struc(t, \Gamma) = \begin{array}{c} \textcircled{t} \end{array}$$

if $t \in TypeVar$ and $\eta_\Gamma(t) = \perp$,

$$struc(t, \Gamma) = struc(\mu t. \alpha, \Gamma)$$

if $t \in TypeVar$ and $\eta_\Gamma(t) = \mu t. \alpha$,

$$struc(B, \Gamma) = \begin{array}{c} \textcircled{B} \end{array}$$

if $B \in BTypes$,

$$struc(\{v\}, \Gamma) = \begin{array}{c} \textcircled{\{ \}} \\ \downarrow \in \\ T \end{array}$$

where $T = struc(v, \Gamma)$,

$$struc(\langle l_1 : v_1, \dots, l_n : v_n \rangle, \Gamma) = \begin{array}{c} \textcircled{\langle \rangle} \\ \swarrow \quad \searrow \\ l_1 \quad \quad l_n \\ \downarrow \quad \quad \downarrow \\ T_1 \quad \dots \quad T_n \end{array}$$

where $T_i = struc(v_i, \Gamma)$,

$$struc(\mu t. \alpha, \Gamma) = struc(\alpha, \Gamma \cup \{\mu t. \alpha\}),$$

where η_Γ is a partial function from $TypeVar$ to $Types$ induced by Γ ; $\eta_\Gamma(t) = \tau$ if τ is a type in Γ and τ starts with μt (there is at most one such type: cf. Lemma 4.18) and $\eta_\Gamma(t) = \perp$ if there is no type in Γ that starts with μt . For convenience, we sometimes write $struc(\tau)$ instead of $struc(\tau, \emptyset)$. \square

The extension of a type is the set of closed terms of which the structure corresponds to the structure of the type.

Definition 4.4 First, for every basic type B , we postulate a disjoint set of constants $Cons_B$, and, for every type variable t , we postulate a disjoint set of instance variables Var_t . The set of all constants, denoted by $Cons$, is given by:

$$Cons = \{ Cons_B \mid B \in BTypes \}.$$

Furthermore, the set of all instance variables, denoted by Var , is given by:

$$Var = \{ Var_t \mid t \in TypeVar \}.$$

Let τ be a type. The set of terms (or instances) of type τ , denoted by $terms(\tau)$, is defined as follows:

$$\begin{aligned} terms(t) &= Var_t \text{ if } t \in TypeVar, \\ terms(B) &= Cons_B \text{ if } B \in BTypes, \\ terms(\{v\}) &= \{ \{e_1, \dots, e_n\} \mid \forall i \in \{1, \dots, n\} [e_i \in terms(v)] \}, \\ terms(< l_1 : v_1, \dots, l_n : v_n >) &= \\ &\quad \{ < l_1 = e_1, \dots, l_n = e_n > \mid \forall i \in \{1, \dots, n\} [e_i \in terms(v_i)] \}, \\ terms(\mu t. \alpha) &= terms(t) \cup \{ \mu x. e_0[x_1 \setminus e_1, \dots, x_n \setminus e_n] \mid \\ &\quad x \in Var_t \wedge e_0 \in terms(\alpha) \wedge \forall i \in \{1, \dots, n\} [\\ &\quad x_i \in Var_t \wedge e_i \in terms(\mu t. \alpha) \wedge x \notin BV(e_i)] \}, \end{aligned}$$

where $BV(e)$ is the set of bound variables in term e :

$$\begin{aligned} BV(y) &= \emptyset \text{ if } y \in Var, \\ BV(b) &= \emptyset \text{ if } b \in Cons, \\ BV(\{e_1, \dots, e_n\}) &= BV(< l_1 = e_1, \dots, l_n = e_n >) = BV(e_1) \cup \dots \cup BV(e_n), \\ BV(\mu y. e) &= BV(e) \cup \{y\}. \end{aligned}$$

The set of all terms, denoted by $Terms$, is given by:

$$Terms = \{ terms(\tau) \mid \tau \in Types \}.$$

Finally, the extension of type τ , denoted by $ext(\tau)$, is defined as:

$$ext(\tau) = \{ e \in terms(\tau) \mid FV(e) \subseteq \{y \in Var_s \mid s \in fvars(\tau)\} \}.$$

where $FV(e)$ is the set of free variables in term e :

$$\begin{aligned}
FV(y) &= \{y\} \text{ if } y \in Var, \\
FV(b) &= \emptyset \text{ if } b \in Cons, \\
FV(\{e_1, \dots, e_n\}) &= FV(\langle l_1 = e_1, \dots, l_n = e_n \rangle) = FV(e_1) \cup \dots \cup FV(e_n), \\
FV(\mu y.e) &= FV(e) - \{y\}.
\end{aligned}$$

□

Example 4.5 Let τ be type $\mu t. \langle a : \text{integer}, b : t \rangle$. Furthermore, let x and y be elements of Var_t . Then $\mu x. \langle a = 1, b = \mu y. \langle a = 2, b = x \rangle \rangle$ is a term of type τ . □

Equality of terms is defined in the same way as equality of types.

Definition 4.6 Let e_1 and e_2 be terms. Strict equality of e_1 and e_2 , denoted by $e_1 = e_2$, is inductively defined as follows:

1. $x = x$ if $x \in Var$
2. $b = b$ if $b \in Cons$
3. $\{e_1, \dots, e_n\} = \{e'_1, \dots, e'_n\}$
if $\exists h : \{1, \dots, n\} \hookrightarrow \{1, \dots, n\} \forall i \in \{1, \dots, n\} [e_i = e'_{h(i)}]$
4. $\langle l_1 = e_1, \dots, l_n = e_n \rangle = \langle l'_1 = e'_1, \dots, l'_n = e'_n \rangle$
if $\forall i \in \{1, \dots, n\} \exists j \in \{1, \dots, n\} [l_i = l'_j \wedge e_i = e'_j]$
5. $\mu x.e = \mu x'.e'$
if $x = x'$ and $e = e'$,

where $h : \{1, \dots, n\} \hookrightarrow \{1, \dots, n\}$ denotes a bijective function. □

Similar to types, terms can be represented by trees.

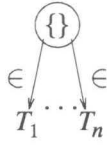
Definition 4.7 Let e be a term of an arbitrary type. The tree representing e is defined as $struc(e, \emptyset)$, where $struc(e, V)$ is defined as:

$$\begin{aligned}
struc(x, V) &= \begin{pmatrix} x \end{pmatrix} \\
&\text{if } x \in Var \text{ and } \eta_V(x) = \perp,
\end{aligned}$$

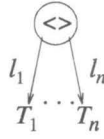
$$\begin{aligned}
struc(x, V) &= struc(\mu x.e_x, V) \\
&\text{if } x \in Var \text{ and } \eta_V(x) = \mu x.e_x,
\end{aligned}$$

$$\begin{aligned}
struc(b, V) &= \begin{pmatrix} b \end{pmatrix} \\
&\text{if } b \in Cons,
\end{aligned}$$

$$struc(\emptyset, V) = \begin{pmatrix} \{\} \end{pmatrix}$$

$$\text{struc}(\{e_1, \dots, e_n\}, V) =$$


where $T_i = \text{struc}(e_i, V)$,

$$\text{struc}(\langle l_1 = e_1, \dots, l_n = e_n \rangle, V) =$$


where $T_i = \text{struc}(e_i, V)$,

$$\text{struc}(\mu x.e_x, V) = \text{struc}(e_x, V \cup \{\mu x.e_x\}),$$

where η_V is a partial function from Var to $Terms$ induced by V ; $\eta_V(x) = e$ if e is a term in V and e starts with μx (there is at most one such term: cf. Lemma 4.18) and $\eta_V(x) = \perp$ if there is no term in V that starts with μx . For convenience, we sometimes write $\text{struc}(e)$ instead of $\text{struc}(e, \emptyset)$. \square

The equivalence relation on terms is defined as an equality relation on the trees of the terms.

Definition 4.8 Let e_1 and e_2 be terms of arbitrary types. Equivalence of e_1 and e_2 , denoted by $e_1 \cong_{TERM} e_2$, is defined by:

$$e_1 \cong_{TERM} e_2 \Leftrightarrow \text{struc}(e_1) = \text{struc}(e_2).$$

Let E_1 and E_2 be sets of terms. Then E_1 is equivalent to E_2 , denoted by $E_1 \cong_{TERMS} E_2$, if and only if:

1. $\forall e_1 \in E_1 \exists e_2 \in E_2 [e_1 \cong_{TERM} e_2]$
2. $\forall e_2 \in E_2 \exists e_1 \in E_1 [e_2 \cong_{TERM} e_1]$.

\square

The following lemma gives the relationship between equivalence of ‘sets of terms’ and equality of ‘sets of trees’.

Lemma 4.9 Let E_1 and E_2 be sets of terms. Then:

$$E_1 \cong_{TERMS} E_2 \Leftrightarrow \{\text{struc}(e) \mid e \in E_1\} = \{\text{struc}(e) \mid e \in E_2\}.$$

Proof. The lemma follows directly from Definition 4.8. \square

For the definition of the subinstance relation, we need the following definition.

Definition 4.10 Let T_1 and T_2 be labeled trees. Furthermore, let φ be a graph homomorphism from T_1 to T_2 . Then φ is a tree morphism if and only if φ maps the root to the root. And T_1 is a supertree of T_2 , denoted by $T_1 \supseteq T_2$, if and only if there is an injective tree morphism from T_2 to T_1 that preserves labels.

Let S_1 and S_2 be sets of labeled trees. Then S_1 is a set of supertrees of S_2 , denoted by $S_1 \supseteq S_2$, if and only if:

$$\forall s_1 \in S_1 \exists s_2 \in S_2 [s_1 \supseteq s_2].$$

□

The subinstance relation on instances is defined as a supertree relation on the trees of the instances.

Definition 4.11 Let e_1 and e_2 be terms of arbitrary types. The subinstance relation, where $e_1 \preceq_{TERM} e_2$ denotes that e_1 is a subinstance of e_2 , is defined by:

$$e_1 \preceq_{TERM} e_2 \Leftrightarrow struc(e_1) \supseteq struc(e_2).$$

Let E_1 and E_2 be sets of terms. Then E_1 is a set of subinstances of E_2 , denoted by $E_1 \preceq_{TERMS} E_2$, if and only if:

$$\forall e_1 \in E_1 \exists e_2 \in E_2 [e_1 \preceq_{TERM} e_2].$$

□

The following lemma gives the relationship between the ‘set of subinstances’ relation and the ‘set of supertrees’ relation.

Lemma 4.12 Let E_1 and E_2 be sets of terms. Then:

$$E_1 \preceq_{TERMS} E_2 \Leftrightarrow \{struc(e) \mid e \in E_1\} \supseteq \{struc(e) \mid e \in E_2\}.$$

Proof. The lemma follows directly from Definition 4.10 and Definition 4.11. □

4.2 Type equivalence

In this section, we define two notions of semantic type equivalence and show that they are interchangeable and decidable using a derivation system. Furthermore, we define a normalisation process for types and prove that the equivalence relation induced by the normalisation process is interchangeable with both notions of semantic type equivalence.

In the previous section, the semantics of a type was defined in terms of a tree representing the structure of the type and in terms of a set of closed terms of which the structure corresponds to the structure of the type. These two notions of type semantics can be used to define two notions of semantic type equivalence: structural and extensional. Structural type equivalence is defined as an equality relation on the trees of the types.

Definition 4.13 Let τ_1 and τ_2 be types. Structural equivalence of τ_1 and τ_2 , denoted by $\tau_1 \cong_{\text{struct}} \tau_2$, is defined as:

$$\tau_1 \cong_{\text{struct}} \tau_2 \Leftrightarrow \text{struct}(\tau_1) = \text{struct}(\tau_2).$$

□

Extensional type equivalence is defined as an equivalence relation on the extensions of the types.

Definition 4.14 Let τ_1 and τ_2 be types. Extensional equivalence of τ_1 and τ_2 , denoted by $\tau_1 \cong_{\text{ext}} \tau_2$, is defined as:

$$\tau_1 \cong_{\text{ext}} \tau_2 \Leftrightarrow \text{ext}(\tau_1) \cong_{\text{TERMS}} \text{ext}(\tau_2).$$

□

4.2.1 Derivation system and normalisation process

In this subsection, we introduce a derivation system for type equivalence and a normalisation process for types. Informally, a derivation is a tree of formulas, where the children formulas imply the parent formula. A formula is of the form $\Gamma \vdash \tau \cong \sigma$ (where τ and σ are types, \cong is type equivalence, and Γ is a context), saying that $\tau \cong \sigma$ follows from the axioms for basic types and the premises in Γ . Context Γ is a triple $(\Gamma_l, \Gamma_r, \Gamma_p)$, where an element of Γ_l is a type definition of the form $\mu t.\alpha$, saying that every type variable t on the left (i.e., in τ) corresponds to type $\mu t.\alpha$, an element of Γ_r is a type definition of the form $\mu s.\beta$, saying that every type variable s on the right (i.e., in σ) corresponds to type $\mu s.\beta$, and an element of Γ_p is a pair of type variables (t, s) , where t occurs on the left and s occurs on the right, saying that t and s are equivalent (and the types they corresponds to). For convenience, we write $\Gamma \cup \{(\mu t.\alpha, \mu s.\beta), (t, s)\}$ instead of $(\Gamma_l \cup \{\mu t.\alpha\}, \Gamma_r \cup \{\mu s.\beta\}, \Gamma_p \cup \{(t, s)\})$.

Definition 4.15 The derivation system for type equivalence, denoted by DE , is defined as follows. The axioms of the derivation system are:

- | | |
|---|---|
| 1. $\Gamma \vdash B \cong B$ | if $B \in BTypes$ |
| 2. $\Gamma \vdash t \cong s$ | if $(t, s) \in \Gamma_p$ |
| 3. $\Gamma \vdash t \cong \mu s.\beta$ | if $(t, s) \in \Gamma_p \wedge \mu s.\beta \in \Gamma_r$ |
| 4. $\Gamma \vdash \mu t.\alpha \cong s$ | if $(t, s) \in \Gamma_p \wedge \mu t.\alpha \in \Gamma_l$ |
| 5. $\Gamma \vdash \mu t.\alpha \cong \mu s.\beta$ | if $(t, s) \in \Gamma_p \wedge \mu t.\alpha \in \Gamma_l \wedge \mu s.\beta \in \Gamma_r$ |

The rules of the derivation system are:

1.
$$\frac{\Gamma \vdash \mu t.\alpha \cong \mu s.\beta}{\Gamma \vdash t \cong s} \quad \text{if } (t, s) \notin \Gamma_p \wedge \mu t.\alpha \in \Gamma_l \wedge \mu s.\beta \in \Gamma_r$$

2. $\frac{\Gamma \vdash \mu t.\alpha \cong \mu s.\beta}{\Gamma \vdash t \cong \mu s.\beta}$ if $(t, s) \notin \Gamma_p \wedge \mu t.\alpha \in \Gamma_l$
3. $\frac{\Gamma \vdash \mu t.\alpha \cong \mu s.\beta}{\Gamma \vdash \mu t.\alpha \cong s}$ if $(t, s) \notin \Gamma_p \wedge \mu s.\beta \in \Gamma_r$
4. $\frac{\Gamma \vdash \tau \cong \sigma}{\Gamma \vdash \{\tau\} \cong \{\sigma\}}$
5. $\frac{\Gamma \vdash \tau_1 \cong \sigma_1, \dots, \Gamma \vdash \tau_n \cong \sigma_n}{\Gamma \vdash \langle l_1 : \tau_1, \dots, l_n : \tau_n \rangle \cong \langle l_1 : \sigma_1, \dots, l_n : \sigma_n \rangle}$
6. $\frac{\Gamma \cup \{(\mu t.\alpha, \mu s.\beta), (t, s)\} \vdash \alpha \cong \beta}{\Gamma \vdash \mu t.\alpha \cong \mu s.\beta}$ if $(t, s) \notin \Gamma_p$.

Rules 1, 2, and 3 introduce folding of types (going from premise to conclusion) and unfolding of types (going from conclusion to premise). \square

The set of axioms and rules of *DE* is the same as the extended set of rules for $=_A$ from [3]. We need the extended set for structural and extensional completeness.

Example 4.16 For convenience, we define a number of abbreviations:

$$\begin{aligned}
 \alpha &= \langle a_1 : B, a_2 : t, a_3 : t \rangle \\
 \beta &= \langle a_1 : B, a_2 : s, a_3 : \mu s'.\beta' \rangle \\
 \beta' &= \langle a_1 : B, a_2 : s, a_3 : s' \rangle \\
 \Gamma_1 &= \{(\mu t.\alpha, \mu s.\beta), (t, s)\} \\
 \Gamma_2 &= \Gamma_1 \cup \{(\mu t.\alpha, \mu s'.\beta'), (t, s')\}.
 \end{aligned}$$

Using derivation system *DE*, we obtain the following derivation for $\emptyset \vdash \mu t.\alpha \cong \mu s.\beta$:

$$\begin{array}{r}
 \frac{\Gamma_2 \vdash B \cong B, \Gamma_2 \vdash t \cong s, \Gamma_2 \vdash t \cong s'}{\Gamma_2 \vdash \alpha \cong \beta'} \text{ (rule 5)} \\
 \frac{\Gamma_2 \vdash \alpha \cong \beta'}{\Gamma_1 \vdash \mu t.\alpha \cong \mu s'.\beta'} \text{ (rule 6)} \\
 \frac{\Gamma_1 \vdash B \cong B, \Gamma_1 \vdash t \cong s, \Gamma_1 \vdash t \cong \mu s'.\beta'}{\Gamma_1 \vdash \alpha \cong \beta} \text{ (rule 5)} \\
 \frac{\Gamma_1 \vdash \alpha \cong \beta}{\emptyset \vdash \mu t.\alpha \cong \mu s.\beta} \text{ (rule 6)}
 \end{array}$$

\square

Derivable type equivalence for closed μ -complete types is given by the following definition.

Definition 4.17 Let τ_1 and τ_2 be closed μ -complete types. Equivalence of τ_1 and τ_2 according to derivation system DE , denoted by $\tau_1 \cong_D \tau_2$, is defined as follows:

$$\tau_1 \cong_D \tau_2 \Leftrightarrow \emptyset \vdash_{DE} \tau_1 \cong \tau_2,$$

where $\Gamma \vdash_{DE} \tau \cong \sigma$ means that there is a derivation in DE with conclusion $\Gamma \vdash \tau \cong \sigma$. \square

Derivable type equality for closed μ -complete types, denoted by $=_D$, is obtained in the same way, from the subsystem of DE that consists of axioms 1 and 2, and rules 4, 5, and 6. Since there is no folding or unfolding in the subsystem, derivable equality is just equality modulo renaming of type variables.

The context of a formula in a derivation induces a function from the set of free type variables on the left to the set of types and a function from the set of free type variables on the right to the set of types.

Lemma 4.18 Let $\Gamma \vdash \tau \cong \sigma$ be a formula in the derivation of $\emptyset \vdash \tau_1 \cong \tau_2$, where τ_1 and τ_2 are closed types. Then η_{Γ_l} , defined by:

$$\eta_{\Gamma_l}(t) = \mu t. \alpha \text{ if } \mu t. \alpha \in \Gamma_l \text{ for some } \alpha,$$

is a function from $fvars(\tau)$ to $Types$ and η_{Γ_r} , defined by:

$$\eta_{\Gamma_r}(t) = \mu t. \alpha \text{ if } \mu t. \alpha \in \Gamma_r \text{ for some } \alpha,$$

is a function from $fvars(\sigma)$ to $Types$.

Proof. To prove the first part of the lemma, let t be an element of $fvars(\tau)$. It suffices to prove that there is exactly one τ' in Γ_l that starts with μt . Since t is bound by μt in τ_1 and rule 6 is the only rule in which μt is removed, there is at least one type in Γ_l that starts with μt . Furthermore, since every type in Γ_l is a substring of τ_1 (follows from a simple induction) and there is only one substring of τ_1 that starts with μt and is also an element of $Types$, there is at most one type in Γ_l that starts with μt .

The proof of the second part is the same as the proof of the first part. \square

Finally, a rewrite process for types is defined based on folding operations.

Definition 4.19 Let τ be a μ -complete type. The reduced form of τ is defined as $rd(\tau, \emptyset)$, where $rd(\tau', \Gamma)$ is defined as follows:

$$\begin{aligned} rd(t, \Gamma) &= t \text{ if } t \in TypeVar \\ rd(B, \Gamma) &= B \text{ if } B \in BTypes \\ rd(\{v\}, \Gamma) &= \{rd(v, \Gamma)\} \\ rd(< l_1 : v_1, \dots, l_n : \tau_n >, \Gamma) &= < l_1 : rd(v_1, \Gamma), \dots, l_n : rd(v_n, \Gamma) > \\ rd(\mu t. \alpha, \Gamma) &= s \\ &\quad \text{if } \exists \mu s. \beta \in \Gamma [struc(\mu t. \alpha, \Gamma) = struc(\mu s. \beta, \Gamma)] \\ rd(\mu t. \alpha, \Gamma) &= \mu t. (rd(\alpha, \Gamma \cup \{\mu t. \alpha\})) \\ &\quad \text{if } \forall \mu s. \beta \in \Gamma [struc(\mu t. \alpha, \Gamma) \neq struc(\mu s. \beta, \Gamma)]. \end{aligned}$$

For convenience, we sometimes write $rd(\tau)$ instead of $rd(\tau, \emptyset)$. \square

The tree representing the reduced form is the same as the tree representing the original type.

Lemma 4.20 Let τ be a μ -complete type. Then:

$$struc(rd(\tau)) = struc(\tau)$$

Proof. The lemma follows by an induction argument on the structure of τ . \square

Since every type has exactly one reduced form, the rewrite process is a normalisation process and the resulting reduced forms are normal forms. For more details about rewrite systems, see [31]. Furthermore, the rewrite process induces an equivalence relation on types.

Definition 4.21 Let τ_1 and τ_2 be closed μ -complete types. Reducible equivalence of τ_1 and τ_2 , denoted by $\tau_1 \cong_R \tau_2$, is defined as follows:

$$\tau_1 \cong_R \tau_2 \Leftrightarrow rd(\tau_1) =_D rd(\tau_2).$$

\square

In the remainder of this chapter, we will prove Theorems 4.22 through 4.25.

Theorem 4.22 Derivable equivalence is sound and complete w.r.t. structural equivalence. \square

Theorem 4.23 Structural and extensional equivalence are logically equivalent. \square

This means that both notions of semantic type equivalence are interchangeable. Combining Theorems 4.22 and 4.23, we can conclude that derivable equivalence is sound and complete w.r.t. extensional equivalence. This means that if a type is equivalent to another type, then any instance of the first type is equivalent to an instance of the second type. It follows that if types are interchangeable, then their instances are interchangeable as well.

Theorem 4.24 Derivable and reducible equivalence are logically equivalent. \square

Theorem 4.25 Derivable equivalence is decidable. \square

Combining Theorems 4.22 through 4.24, we can conclude that all notions of type equivalence are logically equivalent. This means that it does not matter whether type equivalence is defined using trees, extensions, a derivation system, or a normalisation process. Using Theorem 4.25, we can conclude that all notions of type equivalence are decidable.

For the following chapters, the equivalence relation on types is defined as extensional type equivalence.

Definition 4.26 Let τ_1 and τ_2 be closed types. Equivalence of τ_1 and τ_2 , denoted by $\tau_1 \cong_{TYPE} \tau_2$, is defined by:

$$\tau_1 \cong_{TYPE} \tau_2 \Leftrightarrow \tau_1 \cong_{ext} \tau_2.$$

□

4.2.2 Soundness w.r.t. structural equivalence

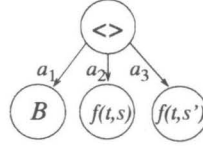
In this subsection, we prove the soundness part of Theorem 4.22. More precisely, for closed μ -complete types τ_1 and τ_2 , we prove:

$$\tau_1 \cong_D \tau_2 \Rightarrow \tau_1 \cong_{struc} \tau_2.$$

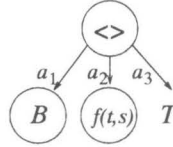
First, for every formula $\Gamma \vdash \tau \cong \sigma$ in the derivation of $\emptyset \vdash \tau_1 \cong \tau_2$, a tree is constructed. The tree is constructed in such a way that it is equal to both $struc(\tau, \Gamma_l)$ and $struc(\sigma, \Gamma_r)$, except for the free type variables. Following the derivation, constructing the tree for a formula from the trees for its children formulas, we obtain a tree that is equal to both $struc(\tau_1, \emptyset)$ and $struc(\tau_2, \emptyset)$, (because τ_1 and τ_2 have no free type variables).

Before giving the definition, we illustrate how to construct the trees corresponding to the formulas in a derivation.

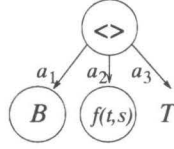
Example 4.27 Let D be the derivation of Example 4.16 and f be an injective function from $\{t\} \times \{s, s'\}$ to $TypeVar - \{t, s, s'\}$. The tree for $\Gamma_2 \vdash \alpha \cong \beta'$ is given by:



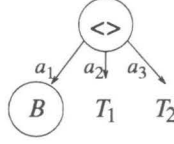
where every subtree corresponds to one of the children formulas of $\Gamma_2 \vdash \alpha \cong \beta'$. The tree for $\Gamma_1 \vdash \mu t. \alpha \cong \mu s'. \beta'$ is given by:



where T is the tree for $\Gamma_1 \vdash \mu t. \alpha \cong \mu s'. \beta'$ itself, resulting in an infinite tree. The resulting tree is an extension of the tree for $\Gamma_2 \vdash \alpha \cong \beta'$. The tree for $\Gamma_1 \vdash t \cong \mu s'. \beta'$ is the same as the tree for $\Gamma_1 \vdash \mu t. \alpha \cong \mu s'. \beta'$ and the tree for $\Gamma_1 \vdash \alpha \cong \beta$ is given by:



where every subtree corresponds to one of the children formulas ($T = \text{tree}(t, \mu s'.\beta', \Gamma_1)$). Finally, the tree for $\emptyset \vdash \mu t.\alpha \cong \mu s.\beta$ is given by:



where T_1 is the tree for $\emptyset \vdash \mu t.\alpha \cong \mu s.\beta$ itself and $T_2 = \text{tree}(t, \mu s'.\beta', \Gamma_1)[f(t, s) \setminus T_1]$. The resulting tree is an extension of the tree for $\Gamma_1 \vdash \alpha \cong \beta$. \square

After illustrating how to construct the trees corresponding to the formulas in a derivation, we give the definition.

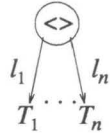
Definition 4.28 Let τ_1 and τ_2 be closed μ -complete types. Furthermore, let f be an injective function from $\text{bvars}(\tau_1) \times \text{bvars}(\tau_2)$ to $\text{TypeVar} - (\text{bvars}(\tau_1) \cup \text{bvars}(\tau_2))$ and $\Gamma \vdash \tau \cong \sigma$ be a formula in the derivation of $\emptyset \vdash \tau_1 \cong \tau_2$. The tree for $\Gamma \vdash \tau \cong \sigma$, denoted by $\text{tree}(\tau, \sigma, \Gamma)$, is defined as follows:

$$\text{tree}(B, B, \Gamma) = \begin{array}{c} \textcircled{B} \\ \text{if } B \in BTypes \end{array}$$

$$\text{tree}(\{\tau'\}, \{\sigma'\}, \Gamma) = \begin{array}{c} \textcircled{\{\}} \\ \downarrow \in \\ T \end{array}$$

where $T = \text{tree}(\tau', \sigma', \Gamma)$

$$\text{tree}(\langle l_1 : \tau_1, \dots, l_n : \tau_n \rangle, \langle l_1 : \sigma_1, \dots, l_n : \sigma_n \rangle, \Gamma) =$$



$$\text{where } T_i = \text{tree}(\tau_i, \sigma_i, \Gamma)$$

$$tree(t, s, \Gamma) = tree(t, \mu s.\beta, \Gamma) = tree(\mu t.\alpha, s, \Gamma) = tree(\mu t.\alpha, \mu s.\beta, \Gamma) =$$

$$\textcircled{f(t,s)}$$

if $(t, s) \in \Gamma_p$

$$tree(t, s, \Gamma) = tree(\mu t.\alpha, \mu s.\beta, \Gamma) \\ \text{if } (t, s) \notin \Gamma_p \wedge \eta_{\Gamma_l}(t) = \mu t.\alpha \wedge \eta_{\Gamma_r}(s) = \mu s.\beta$$

$$tree(t, \mu s.\beta, \Gamma) = tree(\mu t.\alpha, \mu s.\beta, \Gamma) \\ \text{if } (t, s) \notin \Gamma_p \wedge \eta_{\Gamma_l}(t) = \mu t.\alpha$$

$$tree(\mu t.\alpha, s, \Gamma) = tree(\mu t.\alpha, \mu s.\beta, \Gamma) \\ \text{if } (t, s) \notin \Gamma_p \wedge \eta_{\Gamma_r}(s) = \mu s.\beta$$

$$tree(\mu t.\alpha, \mu s.\beta, \Gamma) = tree(\alpha, \beta, \Gamma \cup \{(\mu t.\alpha, \mu s.\beta)\}) \\ \text{if } (t, s) \notin \Gamma_p.$$

□

Finally, we prove Claim 1:

$$\emptyset \vdash_{DE} \tau_1 \cong \tau_2 \Rightarrow (tree(\tau_1, \tau_2, \emptyset) = struc(\tau_1, \emptyset) \wedge tree(\tau_1, \tau_2, \emptyset) = struc(\tau_2, \emptyset)).$$

From this claim it follows that derivable equivalence is sound w.r.t. structural equivalence.

$$\emptyset \vdash_{DE} \tau_1 \cong \tau_2 \Rightarrow struc(\tau_1) = struc(\tau_2).$$

4.2.2.1 Proof of Claim 1

If $\Gamma \vdash \tau \cong \sigma$ is a formula in the derivation of $\emptyset \vdash \tau_1 \cong \tau_2$, then $tree(\tau, \sigma, \Gamma)$ can contain free type variables, whereas $struc(\tau, \Gamma)$ cannot. Therefore, we prove the following, stronger, version of Claim 1. For every formula $\Gamma \vdash \tau \cong \sigma$ in the derivation tree of $\emptyset \vdash \tau_1 \cong \tau_2$:

$$tree(\tau, \sigma, \Gamma)[f(t', s') \setminus struc(\mu t' . \alpha', \Gamma_l) \mid (t', s') \in \Gamma_p \wedge \eta_{\Gamma_l}(t') = \mu t' . \alpha'] \\ = struc(\tau, \Gamma_l),$$

where $T[t_i \setminus T_i \mid i \in I]$ is the tree obtained from T by replacing every leaf labeled t_i by tree T_i , for every $i \in I$.

The proof is an induction argument on the distance of a formula in the derivation tree to its remotest descendant. For sake of convenience, let $[S_\Gamma]$ denote

$$[f(t', s') \setminus struc(\mu t' . \alpha', \Gamma_l) \mid (t', s') \in \Gamma_p \wedge \eta_{\Gamma_l}(t') = \mu t' . \alpha'].$$

Base step: the formula is an axiom.

Axiom 1: $\Gamma \vdash B \cong B$. Then, obviously:

$$\text{tree}(B, B, \Gamma)[S_\Gamma] = \text{tree}(B, B, \Gamma) = \text{struc}(B, \Gamma_l).$$

Axiom 2: $\Gamma \vdash t \cong s$. Then (t, s) must be an element of Γ_p and there must be a type α , such that $\eta_{\Gamma_l}(t) = \mu t.\alpha$. Hence:

$$\text{tree}(t, s, \Gamma)[S_\Gamma] = \text{struc}(\mu t.\alpha, \Gamma_l) = \text{struc}(t, \Gamma_l).$$

Axiom 3, 4, or 5: $\Gamma \vdash t \cong \mu s.\beta$, $\Gamma \vdash \mu t.\alpha \cong s$, or $\Gamma \vdash \mu t.\alpha \cong \mu s.\beta$. Similar to the previous case.

Induction step: the formula is the result of applying a rule to a number of formulas which are closer to their remotest descendant.

Rule 1: $\Gamma \vdash t \cong s$. Then there must be types $\eta_{\Gamma_l}(t) = \mu t.\alpha$ and $\eta_{\Gamma_r}(s) = \mu s.\beta$, such that:

$$\Gamma \vdash \mu t.\alpha \cong \mu s.\beta.$$

Using the induction hypothesis, we can conclude that:

$$\text{tree}(\mu t.\alpha, \mu s.\beta, \Gamma)[S_\Gamma] = \text{struc}(\mu t.\alpha, \Gamma_l).$$

Since (t, s) is not an element of Γ_p , it follows that:

$$\text{tree}(t, s, \Gamma)[S_\Gamma] = \text{tree}(\mu t.\alpha, \mu s.\beta, \Gamma)[S_\Gamma] = \text{struc}(\mu t.\alpha, \Gamma_l) = \text{struc}(t, \Gamma_l).$$

Rule 2 or 3: $\Gamma \vdash t \cong \mu s.\beta$ or $\Gamma \vdash \mu t.\alpha \cong s$. Similar to the previous case.

Rule 4: $\Gamma \vdash \{\tau\} \cong \{\sigma\}$. From $\Gamma \vdash \tau \cong \sigma$ and the induction hypothesis, it follows that:

$$\text{tree}(\tau, \sigma, \Gamma)[S_\Gamma] = \text{struc}(\tau, \Gamma_l).$$

Hence:

$$\text{tree}(\{\tau\}, \{\sigma\}, \Gamma)[S_\Gamma] = \text{struc}(\{\tau\}, \Gamma_l).$$

Rule 5: $\Gamma \vdash \langle l_1 : \tau_1, \dots, l_n : \tau_n \rangle \cong \langle l_1 : \sigma_1, \dots, l_n : \sigma_n \rangle$. Using the induction hypothesis for $\Gamma \vdash \tau_i \cong \sigma_i$, we can conclude that:

$$\text{tree}(\tau_i, \sigma_i, \Gamma)[S_\Gamma] = \text{struc}(\tau_i, \Gamma_l).$$

Hence:

$$\begin{aligned} \text{tree}(\langle l_1 : \tau_1, \dots, l_n : \tau_n \rangle, \langle l_1 : \sigma_1, \dots, l_n : \sigma_n \rangle, \Gamma)[S_\Gamma] \\ = \text{struc}(\langle l_1 : \tau_1, \dots, l_n : \tau_n \rangle, \Gamma_l). \end{aligned}$$

Rule 6: $\Gamma \vdash \mu t.\alpha \cong \mu s.\beta$. Let Γ' be $\Gamma \cup \{(\mu t.\alpha, \mu s.\beta)\}$ and Δ be $\Gamma' \cup \{(t, s)\}$. From $\Delta \vdash \alpha \cong \beta$ and the induction hypothesis, it follows that:

$$\text{a) } tree(\alpha, \beta, \Delta)[S_\Delta] = struc(\alpha, \Delta_l).$$

The proof is based on a) and the fact that $tree(\mu t.\alpha, \mu s.\beta, \Gamma)$ is an infinite repetition of $tree(\alpha, \beta, \Delta)$. To formalise the infinite repetition of $tree(\alpha, \beta, \Delta)$, we define $tree_i(\alpha, \beta, \Delta)$ for $i \in \mathbb{N}$:

$$\begin{aligned} tree_1(\alpha, \beta, \Delta) &= tree(\alpha, \beta, \Delta), \\ tree_{i+1}(\alpha, \beta, \Delta) &= tree(\alpha, \beta, \Delta)[f(t, s) \setminus tree_i(\alpha, \beta, \Delta)]. \end{aligned}$$

Then, for every $i \in \mathbb{N}$, we have the following two properties (which will be proven):

$$\begin{aligned} \text{b) } tree_i(\alpha, \beta, \Delta)[f(t, s) \setminus tree(\alpha, \beta, \Gamma')] &= tree(\alpha, \beta, \Gamma') \\ \text{c) } tree_i(\alpha, \beta, \Delta)[S_\Delta] &= struc(\alpha, \Delta_l). \end{aligned}$$

Property b) states that $tree_i(\alpha, \beta, \Delta)$ is equal to $tree(\alpha, \beta, \Gamma')$ from the root to at least depth i . Hence, $tree(\alpha, \beta, \Gamma')[S_\Delta]$ is equal to $tree_i(\alpha, \beta, \Delta)[S_\Delta]$ from the root to at least depth i . Furthermore, from the fact that $f(t, s)$ does not appear in $tree(\alpha, \beta, \Gamma')$, it follows that:

$$tree(\alpha, \beta, \Gamma')[S_\Gamma] = tree(\alpha, \beta, \Gamma')[S_\Delta].$$

Using property c), we can deduce that $tree(\alpha, \beta, \Gamma')[S_\Gamma]$ is equal to $struc(\alpha, \Delta_l)$. Hence:

$$tree(\mu t.\alpha, \mu s.\beta, \Gamma)[S_\Gamma] = tree(\alpha, \beta, \Gamma')[S_\Gamma] = struc(\alpha, \Delta_l) = struc(\mu t.\alpha, \Gamma_l).$$

We prove b) by an induction argument on i . The base step is an induction argument on the distance of a formula $\Delta' \vdash \tau \cong \sigma$ to its remotest descendant in the derivation tree for $\Delta \vdash \alpha \cong \beta$ to prove that:

$$tree(\tau, \sigma, \Delta')[f(t, s) \setminus tree(\alpha, \beta, \Gamma')] = tree(\tau, \sigma, \Gamma').$$

The non-trivial case of this induction argument is :

$$\begin{aligned} tree(t, s, \Delta')[f(t, s) \setminus tree(\alpha, \beta, \Gamma')] &= \\ tree(\alpha, \beta, \Gamma') &= tree(\mu t.\alpha, \mu s.\beta, \Gamma') = tree(t, s, \Gamma'). \end{aligned}$$

The induction step of the first induction argument is:

$$\begin{aligned} tree_{i+1}(\alpha, \beta, \Delta)[f(t, s) \setminus tree(\alpha, \beta, \Gamma')] &= \\ (tree(\alpha, \beta, \Delta)[f(t, s) \setminus tree_i(\alpha, \beta, \Delta)] &[f(t, s) \setminus tree(\alpha, \beta, \Gamma')]) = \\ tree(\alpha, \beta, \Delta)[f(t, s) \setminus (tree_i(\alpha, \beta, \Delta) &[f(t, s) \setminus tree(\alpha, \beta, \Gamma')])] = \\ tree(\alpha, \beta, \Delta)[f(t, s) \setminus tree(\alpha, \beta, \Gamma')] &= \\ tree(\alpha, \beta, \Gamma'), \end{aligned}$$

where the first step follows from the definition of $tree_{i+1}$, the second from the definition of substitution, the third from the induction hypothesis, and the fourth from the base step.

We prove c) by an induction argument on i . The base step follows from a) and the induction step is:

$$\begin{aligned}
 tree_{i+1}(\alpha, \beta, \Delta)[S_\Delta] &= (tree(\alpha, \beta, \Delta)[f(t, s) \setminus tree_i(\alpha, \beta, \Delta)])(S_\Delta) = \\
 &= (tree(\alpha, \beta, \Delta)[f(t, s) \setminus (tree_i(\alpha, \beta, \Delta)[S_\Delta]), S_\Gamma] = \\
 &= (tree(\alpha, \beta, \Delta)[f(t, s) \setminus struc(\alpha, \Delta_l), S_\Gamma] = \\
 &= (tree(\alpha, \beta, \Delta)[f(t, s) \setminus struc(\mu t. \alpha, \Delta_l), S_\Gamma] = \\
 &= tree(\alpha, \beta, \Delta)[S_\Delta] = \\
 &= struc(\alpha, \Delta_l),
 \end{aligned}$$

where the first step follows from the definition of $tree_{i+1}$, the second follows from the definition of substitution and the fact that $\Delta_p = \Gamma_p \cup \{(t, s)\}$, the third from the induction hypothesis, the fourth from the definition of $struc$, the fifth from the definition of S_Δ , and the final from a).

In the same way, we can prove the following claim; for every formula $\Gamma \vdash \tau \cong \sigma$ in the derivation tree of $\emptyset \vdash \tau_1 \cong \tau_2$:

$$\begin{aligned}
 tree(\tau, \sigma, \Gamma)[f(t', s') \setminus struc(\mu s'. \beta', \Gamma_r) \mid (t', s') \in \Gamma \wedge \eta_{\Gamma_r}(s') = \mu s'. \beta'] \\
 = struc(\sigma, \Gamma_r).
 \end{aligned}$$

4.2.3 Completeness w.r.t. structural equivalence

In this subsection, we prove the completeness part of Theorem 4.22. More precisely, for closed μ -complete types τ_1 and τ_2 , we prove:

$$\tau_1 \cong_{struc} \tau_2 \Rightarrow \tau_1 \cong_D \tau_2.$$

First, a structural equivalence tree for τ_1 and τ_2 is constructed (the structural equivalence tree will be proven isomorphic to the derivation tree with conclusion $\emptyset \vdash \tau_1 \cong \tau_2$). The tree is constructed in such a way that every node is labeled by a tuple of the form (τ, σ, Γ) , where τ and σ are obtained by using the structure of τ_1 and τ_2 (and, if necessary, by unfolding types), such that $struc(\tau, \Gamma_l) = struc(\sigma, \Gamma_r)$. For example, the root is labeled $(\tau_1, \tau_2, \emptyset)$.

For the definition of structural equivalence trees, we need the following lemma.

Lemma 4.29 Let τ and σ be μ -complete types and Γ be a context, such that $struc(\tau, \Gamma_l) = struc(\sigma, \Gamma_r)$ and $struc(\tau, \Gamma_l)$ contains no type variables. Then:

1. if $\tau = B$, then $\sigma = B$
2. if $\tau = \{\tau_1\}$, then $\sigma = \{\sigma_1\}$ and $struc(\tau_1, \Gamma_l) = struc(\sigma_1, \Gamma_r)$
3. if $\tau = < l_1 : \tau_1, \dots, l_n : \tau_n >$, then
 $\sigma = < l_1 : \sigma_1, \dots, l_n : \sigma_n >$ and $struc(\tau_i, \Gamma_l) = struc(\sigma_i, \Gamma_r)$

4. if $\tau = t$, then either
 - a. $\sigma = s$ and $\eta_{\Gamma_l}(t) = \mu t.\alpha$ and $\eta_{\Gamma_r}(s) = \mu s.\beta$ and $struc(\alpha, \Gamma_l) = struc(\beta, \Gamma_r)$
 - b. $\sigma = \mu s.\beta$ and $\eta_{\Gamma_l}(t) = \mu t.\alpha$ and $struc(\alpha, \Gamma_l) = struc(\beta, \Gamma_r \cup \{\sigma\})$
5. if $\tau = \mu t.\alpha$, then either
 - a. $\sigma = s$ and $\eta_{\Gamma_r}(s) = \mu s.\beta$ and $struc(\alpha, \Gamma_l \cup \{\tau\}) = struc(\beta, \Gamma_r)$
 - b. $\sigma = \mu s.\beta$ and $struc(\alpha, \Gamma_l \cup \{\tau\}) = struc(\beta, \Gamma_r \cup \{\sigma\})$.

Proof. The lemma follows directly from Definition 4.3. \square

Now, we can define the children of a node in a structural equivalence tree.

Definition 4.30 Let $x = (\tau, \sigma, \Gamma)$ be a tuple, such that τ and σ are μ -complete types, Γ is a context, $struc(\tau, \Gamma_l) = struc(\sigma, \Gamma_r)$, and $struc(\tau, \Gamma_l)$ contains no type variables. According to Lemma 4.29, there are 5 cases for x , of which the last two cases each have two subcases. For the definition of the children of x , we divide both subcases into two new subcases (one for $(t, s) \in \Gamma_p$ and one for $(t, s) \notin \Gamma_p$), obtaining 11 cases for x . The set of children of x , denoted by $eqchildren(x)$ is defined as follows:

1. if $x = (B, B, \Gamma)$, then $eqchildren(x) = \emptyset$
2. if $x = (t, s, \Gamma)$ and $(t, s) \in \Gamma_p$, then $eqchildren(x) = \emptyset$
3. if $x = (t, \mu s.\beta, \Gamma)$ and $(t, s) \in \Gamma_p$, then $eqchildren(x) = \emptyset$
4. if $x = (\mu t.\alpha, s, \Gamma)$ and $(t, s) \in \Gamma_p$, then $eqchildren(x) = \emptyset$
5. if $x = (\mu t.\alpha, \mu s.\beta, \Gamma)$ and $(t, s) \in \Gamma_p$, then $eqchildren(x) = \emptyset$
6. if $x = (\{\tau\}, \{\sigma\}, \Gamma)$, then $eqchildren(x) = \{(\tau, \sigma, \Gamma)\}$
7. if $x = (\langle l_1 : \tau_1, \dots, l_n : \tau_n \rangle, \langle l_1 : \sigma_1, \dots, l_n : \sigma_n \rangle, \Gamma)$, then $eqchildren(x) = \{(\tau_1, \sigma_1, \Gamma), \dots, (\tau_n, \sigma_n, \Gamma)\}$
8. if $x = (t, s, \Gamma)$ and $(t, s) \notin \Gamma_p \wedge \mu t.\alpha \in \Gamma_l \wedge \mu s.\beta \in \Gamma_r$, then $eqchildren(x) = \{(\mu t.\alpha, \mu s.\beta, \Gamma)\}$
9. if $x = (t, \mu s.\beta, \Gamma)$ and $(t, s) \notin \Gamma_p \wedge \mu t.\alpha \in \Gamma_l$, then $eqchildren(x) = \{(\mu t.\alpha, \mu s.\beta, \Gamma)\}$
10. if $x = (\mu t.\alpha, s, \Gamma)$ and $(t, s) \notin \Gamma_p \wedge \mu s.\beta \in \Gamma_r$, then $eqchildren(x) = \{(\mu t.\alpha, \mu s.\beta, \Gamma)\}$
11. if $x = (\mu t.\alpha, \mu s.\beta, \Gamma)$ and $(t, s) \notin \Gamma_p$, then $eqchildren(x) = \{(\alpha, \beta, \Gamma \cup \{(\mu t.\alpha, \mu s.\beta), (t, s)\})\}$.

\square

Before giving the definition, we give an example of how to construct a structural equivalence tree.

Example 4.31 First, we define the following abbreviations:

$$\alpha = \langle a_1 : B, a_2 : t, a_3 : t \rangle$$

$$\begin{aligned}
\beta &= \langle a_1 : B, a_2 : s, a_3 : \mu s'.\beta' \rangle \\
\beta' &= \langle a_1 : B, a_2 : s, a_3 : s' \rangle \\
\Gamma_1 &= \{(\mu t.\alpha, \mu s.\beta), (t, s)\} \\
\Gamma_2 &= \Gamma_1 \cup \{(\mu t.\alpha, \mu s'.\beta'), (t, s')\}.
\end{aligned}$$

Then $\text{struc}(\mu t.\alpha, \emptyset) = \text{struc}(\mu s.\beta, \emptyset)$. The descendants of $(\mu t.\alpha, \mu s.\beta, \emptyset)$ are given by:

$$\begin{aligned}
\text{eqchildren}((\mu t.\alpha, \mu s.\beta, \emptyset)) &= \{(\alpha, \beta, \Gamma_1)\} \\
\text{eqchildren}((\alpha, \beta, \Gamma_1)) &= \{(B, B, \Gamma_1), (t, s, \Gamma_1), (t, \mu s'.\beta', \Gamma_1)\} \\
\text{eqchildren}((B, B, \Gamma_1)) &= \emptyset \\
\text{eqchildren}((t, s, \Gamma_1)) &= \emptyset \\
\text{eqchildren}((t, \mu s'.\beta', \Gamma_1)) &= \{(\mu t.\alpha, \mu s'.\beta', \Gamma_1)\} \\
\text{eqchildren}((\mu t.\alpha, \mu s'.\beta', \Gamma_1)) &= \{(\alpha, \beta', \Gamma_2)\} \\
\text{eqchildren}((\alpha, \beta', \Gamma_2)) &= \{(B, B, \Gamma_2), (t, s, \Gamma_2), (t, s', \Gamma_2)\} \\
\text{eqchildren}((B, B, \Gamma_2)) &= \emptyset \\
\text{eqchildren}((t, s, \Gamma_2)) &= \emptyset \\
\text{eqchildren}((t, s', \Gamma_2)) &= \emptyset.
\end{aligned}$$

□

The following lemma states that, if a tuple satisfies the precondition of Definition 4.30, then so do its children.

Lemma 4.32 Let $x = (\tau, \sigma, \Gamma)$ be a tuple, such that τ and σ are μ -complete types, Γ is a context, $\text{struc}(\tau, \Gamma_l)$ is equal to $\text{struc}(\sigma, \Gamma_r)$, and $\text{struc}(\tau, \Gamma_l)$ contains no type variables. Then every element of $\text{eqchildren}(x)$ is a tuple $x' = (\tau', \sigma', \Gamma')$, such that $\text{struc}(\tau', \Gamma'_l) = \text{struc}(\sigma', \Gamma'_r)$ and $\text{struc}(\tau', \Gamma'_l)$ contains no type variables.
Proof. The lemma follows from Definition 4.30 and Lemma 4.29. □

Using Lemma 4.32, we can finally give the definition of structural equivalence trees.

Definition 4.33 Let τ_1 and τ_2 be closed μ -complete types, such that $\text{struc}(\tau_1, \emptyset) = \text{struc}(\tau_2, \emptyset)$. The structural equivalence tree for τ_1 and τ_2 is defined as $\text{eqtree}((\tau_1, \tau_2, \emptyset))$, where $\text{eqtree}((\tau, \sigma, \Gamma))$ is defined as follows:

1. if $x = (\tau, \sigma, \Gamma)$ and $\text{eqchildren}(x) = \emptyset$,
then $\text{eqtree}(x)$ has only one node, labeled (τ, σ, Γ)
2. if $x = (\tau, \sigma, \Gamma)$ and $\text{eqchildren}(x) \neq \emptyset$,
then $\text{eqtree}(x)$ consists of a root, labeled (τ, σ, Γ) , and,
for every $y \in \text{eqchildren}(x)$, a subtree $\text{eqtree}(y)$ and an arrow
from the root to subtree $\text{eqtree}(y)$.

□

Lemma 4.34 The structural equivalence tree for τ_1 and τ_2 is finite.

Proof. The structural equivalence tree for τ_1 and τ_2 is constructed by starting with root $(\tau_1, \tau_2, \emptyset)$ and applying the definition of *eqchildren* to the leaves, until every leaf has an empty set of children. Case 6, 7, and 11 can only be applied a finite number of times consecutively, because they decrease the complexity of the types. Case 8, 9, and 10 can only be applied to a leaf (t, s, Γ) (resp., $(t, \mu s.\beta, \Gamma)$ and $(\mu t.\alpha, s, \Gamma)$) if (t, s) is not an element of Γ . However, after one of these rules has been applied, case 11 will be applied, adding (t, s) to Γ . This means that neither case 8, 9, or 10 can be applied more than once for the same pair of type variables (t, s) on a path from the root to a leaf. Since there are only finitely many type variables in τ_1 and τ_2 , case 8, 9, and 10 can only be applied a finite number of times.

From these observations it follows that every rule can only be applied a finite number of times. Hence, the resulting structural equivalence tree is finite. \square

Finally, we prove Claim 2:

for every node labeled (τ, σ, Γ) in $eqtree(\tau_1, \tau_2, \emptyset)$: $\Gamma \vdash_{DE} \tau \cong \sigma$.

From the definition of *eqtree* and Claim 2 it follows that derivable equivalence is complete w.r.t. structural equivalence:

$$struc(\tau_1) = struc(\tau_2) \Rightarrow \emptyset \vdash_{DE} \tau_1 \cong \tau_2.$$

4.2.3.1 Proof of Claim 2

The proof is an induction argument on the distance of a node to its remotest descendant. Base step: the node is a leaf.

Case 1: x is labeled (B, B, Γ) . Then, using axiom 1 of the derivation system, we have: $\Gamma \vdash_{DE} B \cong B$.

Case 2: x is labeled (t, s, Γ) and $(t, s) \in \Gamma_p$. Then, using axiom 2, we have: $\Gamma \vdash_{DE} t \cong s$.

Case 3: x is labeled $(t, \mu s.\beta, \Gamma)$ and $(t, s) \in \Gamma_p$. Then, using axiom 3, we have: $\Gamma \vdash_{DE} t \cong \mu s.\beta$.

Case 4: x is labeled $(\mu t.\alpha, s, \Gamma)$ and $(t, s) \in \Gamma_p$. Then, using axiom 4, we have: $\Gamma \vdash_{DE} \mu t.\alpha \cong s$.

Case 5: x is labeled $(\mu t.\alpha, \mu s.\beta, \Gamma)$ and $(t, s) \in \Gamma_p$. Then, using axiom 5, we have: $\Gamma \vdash_{DE} \mu t.\alpha \cong \mu s.\beta$.

Induction step: the node is the parent of a number of nodes which are closer to their remotest descendant.

Case 6: x is labeled $(\{\tau'\}, \{\sigma'\}, \Gamma)$. The only child of x is labeled (τ', σ', Γ) . Using the induction hypothesis, we can conclude: $\Gamma \vdash_{DE} \tau \cong \sigma$. Hence, from rule 4 of the derivation system, it follows that: $\Gamma \vdash_{DE} \{\tau\} \cong \{\sigma\}$.

Case 7: x is labeled $(\langle l_1 : \tau_1, \dots, l_n : \tau_n \rangle, \langle l_1 : \sigma_1, \dots, l_n : \sigma_n \rangle, \Gamma)$. The

children of x are labeled $(\tau_i, \sigma_i, \Gamma)$. Using the induction hypothesis, for every $i \in \{1, \dots, n\}$ we can conclude: $\Gamma \vdash_{DE} \tau_i \cong \sigma_i$. Hence, from rule 5 of the derivation system, it follows that: $\Gamma \vdash_{DE} \langle l_1 : \tau_1, \dots, l_n : \tau_n \rangle \cong \langle l_1 : \sigma_1, \dots, l_n : \sigma_n \rangle$.

Case 8: x is labeled (t, s, Γ) and $(t, s) \notin \Gamma_p$. The only child of x is labeled $(\mu t. \alpha, \mu s. \beta, \Gamma)$. Using the induction hypothesis, we can conclude: $\Gamma \vdash_{DE} \mu t. \alpha \cong \mu s. \beta$. Hence, from rule 1, it follows that: $\Gamma \vdash_{DE} t \cong s$.

Case 9: x is labeled $(t, \mu s. \beta, \Gamma)$ and $(t, s) \notin \Gamma_p$. The only child of x is labeled $(\mu t. \alpha, \mu s. \beta, \Gamma)$. Using the induction hypothesis, we can conclude: $\Gamma \vdash_{DE} \mu t. \alpha \cong \mu s. \beta$. Hence, from rule 2, it follows that: $\Gamma \vdash_{DE} t \cong \mu s. \beta$.

Case 10: x is labeled $(\mu t. \alpha, s, \Gamma)$ and $(t, s) \notin \Gamma_p$. The only child of x is labeled $(\mu t. \alpha, \mu s. \beta, \Gamma)$. Using the induction hypothesis, we can conclude: $\Gamma \vdash_{DE} \mu t. \alpha \cong \mu s. \beta$. Hence, from rule 3, it follows that: $\Gamma \vdash_{DE} \mu t. \alpha \cong s$.

Case 11: x is labeled $(\mu t. \alpha, \mu s. \beta, \Gamma)$ and $(t, s) \notin \Gamma_p$. The only child of x is labeled $(\alpha, \beta, \Gamma \cup \{\mu t. \alpha, \mu s. \beta, (t, s)\})$. Using the induction hypothesis, we can conclude: $\Gamma \cup \{(\mu t. \alpha, \mu s. \beta), (t, s)\} \vdash_{DE} \alpha \cong \beta$. Hence, from rule 6, it follows that: $\Gamma \vdash_{DE} \mu t. \alpha \cong \mu s. \beta$.

4.2.4 Equivalence of structural and extensional equivalence

In this subsection, we prove Theorem 4.23. More precisely, for closed μ -complete types τ_1 and τ_2 , we prove:

$$\tau_1 \cong_{\text{struc}} \tau_2 \Leftrightarrow \tau_1 \cong_{\text{ext}} \tau_2.$$

First, for every type τ , the structural extension is defined in terms of trees (structural extensions will be proven equal for types represented by the same tree). The structural extension of a type is defined in such a way that it corresponds to the extension of the type.

For the definition of structural instances, we need a number of preliminary definitions. The exact tree of a type is the same as the tree of the type, except that the exact tree contains type variables.

Definition 4.35 Let τ be a type. The exact tree representing τ is defined as $\text{estruc}(\tau, \emptyset)$, where $\text{estruc}(\tau', \Gamma)$ is defined as follows:

$$\begin{aligned} \text{estruc}(t, \Gamma) &= \begin{pmatrix} t \end{pmatrix} \\ &\text{if } t \in \text{TypeVar} \text{ and } \eta_\Gamma(t) = \perp, \end{aligned}$$

$$\begin{aligned} \text{estruc}(t, \Gamma) &= \text{estruc}(\mu t. \alpha, \Gamma) \\ &\text{if } t \in \text{TypeVar} \text{ and } \eta_\Gamma(t) = \mu t. \alpha, \end{aligned}$$

$$\begin{aligned} \text{estruc}(B, \Gamma) &= \begin{pmatrix} B \end{pmatrix} \\ &\text{if } B \in \text{BTypes}, \end{aligned}$$

$$\text{estruc}(\{\tau_1\}, \Gamma) = \begin{array}{c} \textcircled{\{\}} \\ \downarrow \in \\ T \end{array}$$

where $T = \text{estruc}(\tau_1, \Gamma)$,

$$\text{estruc}(\langle l_1 : \tau_1, \dots, l_n : \tau_n \rangle) = \begin{array}{c} \textcircled{\langle \rangle} \\ \swarrow \quad \searrow \\ l_1 \quad l_n \\ \downarrow \quad \downarrow \\ T_1 \quad \dots \quad T_n \end{array}$$

where $T_i = \text{estruc}(\tau_i, \Gamma)$

$$\text{estruc}(\mu t. \langle l_1 : \tau_1, \dots, l_n : \tau_n \rangle, \Gamma) = \begin{array}{c} \textcircled{t} \\ \swarrow \quad \searrow \\ l_1 \quad l_n \\ \downarrow \quad \downarrow \\ T_1 \quad \dots \quad T_n \end{array}$$

where $T_i = \text{estruc}(\tau_i, \Gamma \cup \{\mu t. \langle l_1 : \tau_1, \dots, l_n : \tau_n \rangle\})$.

For convenience, we sometimes write $\text{estruc}(\tau)$ instead of $\text{estruc}(\tau, \emptyset)$. \square

Lemma 4.36 Let τ be a type. Then there is a bijective tree homomorphism φ from $\text{estruc}(\tau)$ to $\text{struc}(\tau)$, such that for every node or arrow q the following holds:

1. if $\text{label}(q) \in \text{TypeVar}$ and q is not a leaf, then $\text{label}(\varphi(q)) = \langle \rangle$
2. otherwise, $\text{label}(\varphi(q)) = \text{label}(q)$.

where $\text{label}(q)$ denotes the label of node or arrow q .

Proof. The lemma follows from Definition 4.3 and Definition 4.35. \square

The exact tree of a term is the same as the tree of the term, except that the exact tree contains instance variables.

Definition 4.37 Let e be a term. The exact tree representing e is defined as $\text{estruc}(e, \emptyset)$, where $\text{estruc}(e', \Gamma)$ is defined as follows:

$$\begin{aligned} \text{estruc}(x, \Gamma) &= \textcircled{x} \\ \text{if } x \in \text{Var} \text{ and } \eta_\Gamma(x) &= \perp, \\ \text{estruc}(x, \Gamma) &= \text{estruc}(\mu x. e_x, \Gamma) \\ \text{if } x \in \text{Var} \text{ and } \eta_\Gamma(x) &= \mu x. e_x, \end{aligned}$$

$$\begin{aligned} \text{estruc}(b, \Gamma) &= \textcircled{b} \\ \text{if } b &\in \text{Cons}, \end{aligned}$$

$$\text{estruc}(\emptyset, \Gamma) = \textcircled{\{\}}$$

$$\text{estruc}(\{e_1, \dots, e_n\}, \Gamma) = \begin{array}{c} \textcircled{\{\}} \\ \downarrow \quad \downarrow \\ \in \quad \dots \quad \in \\ T_1 \quad \dots \quad T_n \end{array}$$

where $T_i = \text{estruc}(e_i, \Gamma)$,

$$\text{estruc}(\langle l_1 = e_1, \dots, l_n = e_n \rangle) = \begin{array}{c} \textcircled{\langle \rangle} \\ \downarrow \quad \downarrow \\ l_1 \quad \dots \quad l_n \\ T_1 \quad \dots \quad T_n \end{array}$$

where $T_i = \text{estruc}(e_i, \Gamma)$

$$\text{estruc}(\mu x. \langle l_1 = e_1, \dots, l_n = e_n \rangle, \Gamma) = \begin{array}{c} \textcircled{x} \\ \downarrow \quad \downarrow \\ l_1 \quad \dots \quad l_n \\ T_1 \quad \dots \quad T_n \end{array}$$

where $T_i = \text{estruc}(e_i, \Gamma \cup \{\mu x. \langle l_1 = e_1, \dots, l_n = e_n \rangle\})$.

For convenience, we sometimes write $\text{estruc}(e)$ instead of $\text{estruc}(e, \emptyset)$. \square

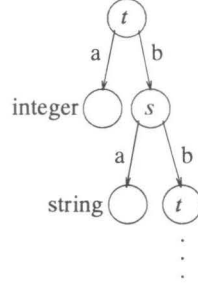
The instance relation between exact trees representing terms and exact trees representing types is defined as follows.

Definition 4.38 Let τ be a type. Then tree T is an instance of $\text{estruc}(\tau)$, denoted by $\text{inst}(T, \text{estruc}(\tau))$, if and only if there is an injective tree morphism from T to $\text{estruc}(\tau)$, such that for every node or arrow q in T the following holds:

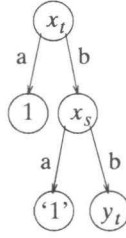
1. if $\text{label}(q) \in \text{Cons}_B$ for some $B \in \text{BTypes}$, then $\text{label}(\varphi(q)) = B$
2. if $\text{label}(q) \in \text{Var}_t$ for some $t \in \text{TypeVar}$ and q is a leaf, then $\text{label}(\varphi(q)) = t$
3. if $\text{label}(q) \in \text{Var}_t$ for some $t \in \text{TypeVar}$ and q is not a leaf, then $\text{label}(\varphi(q)) = t$ and $|\text{children}(q)| = |\text{children}(\varphi(q))|$
4. otherwise, $\text{label}(\varphi(q)) = \text{label}(q)$.

\square

Example 4.39 Let τ be type $\mu t. < a : \text{integer}, b : \mu s. < a : \text{string}, b : t >>$ and e be term $\mu x_t. < a = 1, b = \mu x_s. < a = '1', b = y_t >>$, where x_s is an element of Var_s , and x_t and y_t are elements of Var_t . The exact tree representing τ , denoted by $\text{estruc}(\tau)$, is given by:



The exact tree representing e , denoted by $\text{estruc}(e)$, is given by:



And, obviously, $\text{estruc}(e)$ is an instance of $\text{estruc}(\tau)$. \square

A level-1 subterm of a term is obtained by replacing every set in the term by a subset of cardinality 1.

Definition 4.40 Let e be a term. The set of level-1 subterms of e , denoted by $\text{subs}(e)$, is defined as follows:

$$\begin{aligned}
 \text{subs}(x) &= \{x\} \text{ if } x \in \text{Var}, \\
 \text{subs}(b) &= \{b\} \text{ if } b \in \text{Cons}, \\
 \text{subs}(\emptyset) &= \emptyset, \\
 \text{subs}(\{e_1, \dots, e_n\}) &= \\
 &\quad \{\{e'\} \mid e' \in \text{subs}(e_1)\} \cup \dots \cup \{\{e'\} \mid e' \in \text{subs}(e_n)\}, \\
 \text{subs}(< l_1 = e_1, \dots, l_n = e_n >) &= \\
 &\quad \{< l_1 = e'_1, \dots, l_n = e'_n > \mid \forall i \in \{1, \dots, n\} [e'_i \in \text{subs}(e_i)]\}, \\
 \text{subs}(\mu x. e) &= \{\mu x. e' \mid e' \in \text{subs}(e)\}.
 \end{aligned}$$

\square

Example 4.41 Let e be term $\mu x. < a = \{1, 2\}, b = \mu y. < a = \{2, 5\}, b = x >>$. The set of level-1 subterms of e is given by:

$$\begin{aligned}
& \{\mu x. \langle a = \{1\}, b = \mu y. \langle a = \{2\}, b = x \rangle \rangle \\
& \quad \mu x. \langle a = \{1\}, b = \mu y. \langle a = \{5\}, b = x \rangle \rangle \\
& \quad \mu x. \langle a = \{2\}, b = \mu y. \langle a = \{2\}, b = x \rangle \rangle \\
& \quad \mu x. \langle a = \{2\}, b = \mu y. \langle a = \{5\}, b = x \rangle \rangle\}.
\end{aligned}$$

□

Now we can define structural extensions.

Definition 4.42 Let τ be a type. The structural extension of τ , denoted by $struc_ext(\tau)$, is defined as:

$$struc_ext(\tau) = \{ struc(e) \mid e \in Terms \wedge FV(e) \subseteq \{y \in Var_s \mid s \in fvars(\tau)\} \wedge \forall e' \in subs(e)[inst(estruc(e), estruc(\tau))] \}.$$

□

Note that the elements of the structural extension of a type contain sets of arbitrary cardinality. Furthermore, we define the projection of a term on (the tree of) a type. A term is projected on a type by unfolding the term and renaming instance variables until the resulting term matches the type exactly.

Definition 4.43 Let τ_1 and τ_2 be types, such that $struc(\tau_1) = struc(\tau_2)$. Furthermore, let g be an injective function from $bvars(\tau_1) \times TypeVar$ to $Var - fvars(\tau_1)$ and e be a term, such that $\forall e' \in subs(e)[inst(estruc(e'), estruc(\tau_1))]$. The projection of e on $estruc(\tau_2)$ is defined as $proj(e, estruc(\tau_2), \emptyset)$, where $proj(e', U, V)$ is defined as follows:

$$\begin{aligned}
& proj(x, node(t), V) = x \text{ if } x \in Var \\
& proj(b, node(B), V) = b \text{ if } b \in Cons \\
& proj(\emptyset, tree(\{T\}), V) = \emptyset \\
& proj(\{e_1, \dots, e_n\}, tree(\{T\}), V) = \{proj(e_1, T, V), \dots, proj(e_n, T, V)\} \\
& proj(x, tree(t. \langle l_1 : T_1, \dots, l_n : T_n \rangle), V) = g(x, t) \\
& \quad \text{if } g(x, t) \in V \\
& proj(x, tree(t. \langle l_1 : T_1, \dots, l_n : T_n \rangle), V) = \\
& \quad proj(\eta_V(x), tree(t. \langle l_1 : T_1, \dots, l_n : T_n \rangle), V) \\
& \quad \text{if } g(x, t) \notin V \\
& proj(\mu x. \langle l_1 = e_1, \dots, l_n = e_n \rangle, tree(t. \langle l_1 : T_1, \dots, l_n : T_n \rangle), V) = g(x, t) \\
& \quad \text{if } g(x, t) \in V \\
& proj(\mu x. \langle l_1 = e_1, \dots, l_n = e_n \rangle, tree(t. \langle l_1 : T_1, \dots, l_n : T_n \rangle), V) = \\
& \quad \mu g(x, t). \langle l_1 : proj(e_1, T_1, V \cup \{g(x, t), \mu x. \langle l_1 = e_1, \dots, l_n = e_n \rangle\}), \dots, \\
& \quad \quad l_n : proj(e_n, T_n, V \cup \{g(x, t), \mu x. \langle l_1 = e_1, \dots, l_n = e_n \rangle\}) \rangle \\
& \quad \text{if } g(x, t) \notin V,
\end{aligned}$$

where

$$\text{node}(l) = \textcircled{l}$$

$$\text{tree}(\{T\}) = \textcircled{\{\}} \downarrow \in T$$

$$\text{tree}(l. \langle l_1 : T_1, \dots, l_n : T_n \rangle) = \begin{array}{c} \textcircled{l} \\ \swarrow \quad \searrow \\ l_1 \quad \dots \quad l_n \\ \downarrow \quad \dots \quad \downarrow \\ T_1 \quad \dots \quad T_n \end{array}$$

□

Example 4.44 Let τ be type $\mu t. \langle a : \text{integer}, c : \mu s. \langle a : \text{integer}, b : \text{string}, c : t \rangle \rangle$ and e be term $\mu x. \langle a = 1, b = '1', c = x \rangle$. The projection of e on $\text{estruc}(\tau)$ is given by:

$$\mu g(x, t). \langle a = 1, c = \mu g(x, s). \langle a = 1, b = '1', c = g(x, t) \rangle \rangle.$$

□

Finally, we prove Claim 3:

$$\text{struc}(\tau_1) = \text{struc}(\tau_2) \Leftrightarrow \text{struc_ext}(\tau_1) = \text{struc_ext}(\tau_2),$$

and Claim 4:

$$\text{struc_ext}(\tau) = \{\text{struc}(e) \mid e \in \text{ext}(\tau)\}.$$

From these claims it follows that structural and extensional equivalence are logically equivalent:

$$\begin{aligned} \text{struc}(\tau_1) = \text{struc}(\tau_2) &\Leftrightarrow \\ \text{struc_ext}(\tau_1) = \text{struc_ext}(\tau_2) &\Leftrightarrow \\ \{\text{struc}(e) \mid e \in \text{ext}(\tau_1)\} = \{\text{struc}(e) \mid e \in \text{ext}(\tau_2)\} &\Leftrightarrow \\ \text{ext}(\tau_1) \cong_{\text{TERMS}} \text{ext}(\tau_2). \end{aligned}$$

4.2.4.1 Proof of Claim 3

Proof of \Rightarrow . Suppose $\text{struc}(\tau_1) = \text{struc}(\tau_2)$. Using Lemma 4.36, we can conclude that there is a bijective tree morphism from $\text{estruc}(\tau_1)$ to $\text{estruc}(\tau_2)$, such that for every node or arrow q in $\text{estruc}(\tau_1)$ the following holds:

1. if $\text{label}(q) = t$ for some $t \in \text{Type Var}$ and q is a leaf, then $\text{label}(\varphi(q)) = t$,

2. if $\text{label}(q) = t$ for some $t \in \text{TypeVar}$ and q is not a leaf,
then $\text{label}(\varphi(q)) = s$ for some $s \in \text{TypeVar}$,
3. otherwise, $\text{label}(\varphi(q)) = \text{label}(q)$.

Now, let $\text{struc}(e)$ be an element of $\text{struc_ext}(\tau_1)$ and e' be an element of $\text{subs}(e)$. Furthermore, let $P(e')$ be $\text{proj}(e', \text{estruc}(\tau_2), \emptyset)$. Then $\text{inst}(\text{estruc}(e'), \text{estruc}(\tau_1))$, because e' is an element of $\text{subs}(e)$ and $\text{struc}(e)$ is an element of $\text{struc_ext}(\tau_1)$. From the definition of proj it follows that there is an injective tree morphism from $\text{estruc}(P(e'))$ to $\text{estruc}(\tau_2)$, such that for every node or arrow q in $\text{estruc}(P(e'))$ the following holds:

1. if $\text{label}(q) \in \text{Cons}_B$ for some $B \in \text{BTypes}$,
then $\text{label}(\varphi(q)) = B$
2. if $\text{label}(q) \in \text{Var}_t$ for some $t \in \text{TypeVar}$ and q is a leaf,
then $\text{label}(\varphi(q)) = t$
3. if $\text{label}(q) \in \text{Var}_t$ for some $t \in \text{TypeVar}$ and q is not a leaf,
then $\text{label}(\varphi(q)) = t$ and $|\text{children}(q)| = |\text{children}(\varphi(q))|$
4. otherwise, $\text{label}(\varphi(q)) = \text{label}(q)$.

That is, $\text{inst}(\text{estruc}(P(e')), \text{estruc}(\tau_2))$. Hence, for every $e' \in \text{subs}(e)$, we have:

$$\text{inst}(\text{estruc}(P(e')), \text{estruc}(\tau_2)).$$

Let $P(e)$ be $\text{proj}(e, \text{estruc}(\tau_2), \emptyset)$. Using the definition of proj , we can conclude that $\text{struc}(P(e)) = \text{struc}(e)$ and:

$$\forall e'' \in \text{subs}(P(e)) [\text{inst}(\text{estruc}(e''), \text{estruc}(\tau_2))],$$

because $e'' \in \text{subs}(P(e)) \Leftrightarrow (e'' = P(e') \wedge e' \in \text{subs}(e))$. Since $FV(e) = FV(P(e))$, we have $\text{struc}(e) = \text{struc}(P(e)) \in \text{struc_ext}(\tau_2)$. Hence, $\text{struc_ext}(\tau_1) = \text{struc_ext}(\tau_2)$.

Proof of \Leftarrow . Suppose $\text{struc_ext}(\tau_1) = \text{struc_ext}(\tau_2)$. Let e_1 be a term, such that there is a bijective tree morphism from $\text{estruc}(e_1)$ to $\text{estruc}(\tau_1)$ that satisfies the requirements of Definition 4.38. It follows that there is a term e_2 , such that $\text{struc}(e_1) = \text{struc}(e_2)$ and $\text{inst}(\text{estruc}(e_2), \text{estruc}(\tau_2))$. That is, there is a bijective tree morphism from $\text{estruc}(e_1)$ to $\text{estruc}(e_2)$ that preserves all labels, except type variables, and there is an injective tree morphism from $\text{estruc}(e_2)$ to $\text{estruc}(\tau_2)$ that satisfies the requirements of Definition 4.38. From the fact that the tree morphism from $\text{estruc}(e_1)$ to $\text{estruc}(\tau_1)$ is bijective, it follows that every set in e_1 (and e_2) is a singleton. Hence, the tree morphism from $\text{estruc}(e_2)$ to $\text{estruc}(\tau_2)$ is bijective. Using Lemma 4.36, we can deduce that $\text{struc}(\tau_1) = \text{struc}(\tau_2)$.

4.2.4.2 Proof of Claim 4

To prove $\{struc(e) \mid e \in ext(\tau)\} = struc_ext(\tau)$, where

$$ext(\tau) = \{e \mid e \in terms(\tau) \wedge FV(e) \subseteq \{y \in Var_s \mid s \in fvars(\tau)\}\}$$

and

$$struc_ext(\tau) = \{struc(e) \mid e \in Terms \wedge FV(e) \subseteq \{y \in Var_s \mid s \in fvars(\tau)\} \wedge \forall e' \in subs(e)[inst(estruc(e'), estruc(\tau))]\},$$

it suffices to prove the following claim:

$$e \in terms(\tau) \Leftrightarrow (e \in Terms \wedge \forall e' \in subs(e) [inst(estruc(e'), estruc(\tau))]).$$

The proof is an induction argument on the structure of τ . Let τ be basic type B . Since $inst(estruc(e), estruc(B)) \Leftrightarrow e \in Cons_B$, and, for $b \in Cons_B$, $subs(b) = \{b\}$, we have:

$$b \in terms(B) \Leftrightarrow (b \in Terms \wedge \forall b' \in subs(b) [inst(estruc(b'), estruc(B))]).$$

Let τ be type variable t . Since $inst(estruc(e), estruc(t)) \Leftrightarrow e \in Var_t$, and, for $x \in Var_t$, $subs(x) = \{x\}$, we have:

$$x \in terms(t) \Leftrightarrow (x \in Terms \wedge \forall x' \in subs(x) [inst(estruc(x'), estruc(t))]).$$

Let τ be set type $\{\tau_1\}$. Apply the induction hypothesis to τ_1 and use $subs(\{e_1, \dots, e_n\}) = subs(\{e_1\}) \cup \dots \cup subs(\{e_n\})$.

Let τ be record type $\langle l_1 : \tau_1, \dots, l_n : \tau_n \rangle$. Apply the induction hypothesis to τ_i , for $i \in \{1, \dots, n\}$.

Let τ be recursive type $\mu t. \alpha$. The proof of \Rightarrow is an induction argument on the structure of term $e \in terms(\tau)$.

Base step: $e = x \in Var_t$. Then:

$$\forall x' \in subs(x) [inst(estruc(x'), estruc(\mu t. \alpha))].$$

Induction step: $e = \mu x. (e_0[x_1 \setminus e_1, \dots, x_n \setminus e_n])$, such that $e_0 \in terms(\alpha)$ and $e_i \in terms(\mu t. \alpha)$, for $i \in \{1, \dots, n\}$. Applying the first induction hypothesis to e_0 and the second induction hypothesis to e_1 through e_n , gives us:

1. $\forall e' \in subs(e_0) [inst(estruc(e'), estruc(\alpha))]$
2. $\forall i \in \{1, \dots, n\} \forall e' \in subs(e_i) [inst(estruc(e'), estruc(\mu t. \alpha))].$

Let R_t be the transformation on trees that replaces the label of the root by t . Since

$$subs(e_0[x_1 \setminus e_1, \dots, x_n \setminus e_n]) = \{e'_0[x_1 \setminus e'_1, \dots, x_n \setminus e'_n] \mid e'_0 \in subs(e_0) \wedge \forall i \in \{1, \dots, n\} [e'_i \in subs(e_i)]\},$$

and $R_t(estruc(\alpha))[t \setminus estruc(\mu t.\alpha)] = estruc(\mu t.\alpha)$, we have:

1. $\forall e' \in \text{subs}(e_0[x_1 \setminus e_1, \dots, x_n \setminus e_n])$
 $[inst(estruc(e'), estruc(\alpha)[t \setminus estruc(\mu t.\alpha)])]$
2. $\forall e' \in \text{subs}(\mu x.(e_0[x_1 \setminus e_1, \dots, x_n \setminus e_n])) [inst(estruc(e'), estruc(\mu t.\alpha))]$.

The proof of \Leftarrow is an induction argument on the number of bound instance variables from Var_t in $p_estruc(e)$.

Base step: $estruc(e)$ has no bound instance variables from Var_t . Then $e = x \in Var_t$ and, hence, $e \in \text{terms}(\mu t.\alpha)$.

Induction step: $estruc(e)$ has $j + 1$ bound instance variables from Var_t . Then $e = \mu x.e_x$ and:

$$\forall \mu x.e' \in \text{subs}(\mu x.e_x) [inst(estruc(\mu x.e'), estruc(\mu t.\alpha))].$$

Let R_x be the transformation on trees that replaces the label of the root by x . Since $estruc(\mu x.e') = R_x(estruc(e'))[x \setminus estruc(\mu x.e')]$ and $estruc(\mu t.\alpha) = R_t(estruc(\alpha))[t \setminus estruc(\mu t.\alpha)]$, we have:

$$\forall e' \in \text{subs}(e_x) [inst(estruc(e'), estruc(\alpha)[t \setminus estruc(\mu t.\alpha)])].$$

In fact, we have: $e_x = e_0[x_1 \setminus e_1, \dots, x_n \setminus e_n]$, where $\{x_1, \dots, x_n\} \subset Var_t$, $\{e_0, \dots, e_n\} \subset \text{Terms}$, and:

1. $\forall e' \in \text{subs}(e_0) [inst(estruc(e'), estruc(\alpha))]$
2. $\forall i \in \{1, \dots, n\} \forall e' \in \text{subs}(e_i) [inst(estruc(e'), estruc(\mu t.\alpha))]$.

Applying the first induction hypothesis to e_0 and the second induction hypothesis to e_1 through e_n (every $estruc(e_i)$ has at most j bound instance variables from Var_t), gives us:

$$e_0 \in \text{terms}(\alpha) \wedge \forall i \in \{1, \dots, n\} [e_i \in \text{terms}(\mu t.\alpha)].$$

Hence, $e = \mu x.e_x = \mu x.(e_0[x_1 \setminus e_1, \dots, x_n \setminus e_n]) \in \text{terms}(\mu t.\alpha)$.

4.2.5 Equivalence of derivable and reducible equivalence

In this subsection, we prove Theorem 4.24. More precisely, for closed μ -complete types τ_1 and τ_2 , we prove:

$$\tau_1 \cong_D \tau_2 \Leftrightarrow \tau_1 \cong_R \tau_2.$$

Proof of \Rightarrow . Suppose $\tau_1 \cong_D \tau_2$. Then $struc(\tau_1) = struc(\tau_2)$. Hence, there is a bijective tree morphism from $estruc(\tau_1)$ to $estruc(\tau_2)$, such that for every node or arrow q in $estruc(\tau_1)$ the following holds:

1. if $label(q) = t$ for some $t \in \text{TypeVar}$, then $label(\varphi(q)) = s$ for some $s \in \text{TypeVar}$,

2. otherwise, $\text{label}(\varphi(q)) = \text{label}(q)$.

From the definition of rd , it follows that if τ is a type, φ is the bijective tree homomorphism from $\text{estruc}(rd(\tau))$ to $\text{struc}(rd(\tau))$ as given by Lemma 4.36, n_1 and n_2 are nodes on the same path starting at the root of $\text{estruc}(rd(\tau))$, and the tree starting at $\varphi(n_1)$ is equal to the tree starting at $\varphi(n_2)$, then the label of n_1 is equal to the label of n_2 . Hence, there is a bijective tree morphism from $\text{estruc}(rd(\tau_1))$ to $\text{estruc}(rd(\tau_2))$ and a bijective function f from $\text{bvars}(rd(\tau_1))$ to $\text{bvars}(rd(\tau_2))$, such that for every node or arrow q in $\text{estruc}(\tau_1)$ the following holds:

1. if $\text{label}(q) = t$ for some $t \in \text{TypeVar}$, then $\text{label}(\varphi(q)) = f(t)$
2. otherwise, $\text{label}(\varphi(q)) = \text{label}(q)$.

That is, $rd(\tau_1)$ and $rd(\tau_2)$ are equal, modulo renaming of type variables. Hence, $rd(\tau_1) =_D rd(\tau_2)$.

Proof of \Leftarrow . Suppose $rd(\tau_1) =_D rd(\tau_2)$. That is, $rd(\tau_1)$ and $rd(\tau_2)$ are equal, modulo renaming of type variables. Then:

$$\text{struc}(\tau_1) = \text{struc}(rd(\tau_1)) = \text{struc}(rd(\tau_2)) = \text{struc}(\tau_2)$$

Hence, $\tau_1 \cong_D \tau_2$.

4.2.6 Decidability of derivable equivalence

In this subsection, we prove Theorem 4.25. More precisely, we prove that there is a decision procedure for derivable equivalence.

Suppose τ_1 and τ_2 are closed μ -complete types, and we want to know whether $\tau_1 \cong_D \tau_2$ or $\tau_1 \not\cong_D \tau_2$. Then we try to derive $\emptyset \vdash \tau_1 \cong \tau_2$ bottom up, by starting from $\emptyset \vdash \tau_1 \cong \tau_2$ and by applying the rules of derivation system DE until no rules can be applied any more. For every formula of the form $\Gamma \vdash \tau \cong \sigma$, at most one rule can be applied. Hence, the derivation process is deterministic: there is at most one derivation.

Rule 4, 5, and 6 can only be applied a finite number of times consecutively, because they decrease the complexity of the types. Rule 1, 2, and 3 can only be applied to a formula $\Gamma \vdash t \cong s$ (resp., $\Gamma \vdash t \cong \mu s.\beta$ and $\Gamma \vdash \mu t.\alpha \cong s$) if (t, s) is not an element of Γ . However, after one of these rules has been applied, rule 6 will be applied, adding (t, s) to Γ . This means that neither rule 1, 2, or 3 can be applied more than once for the same pair of type variables (t, s) on a path from the root to a leaf. Since there are only finitely many type variables in τ_1 and τ_2 , rule 1, 2, and 3 can only be applied a finite number of times.

From these observations it follows that every rule can only be applied a finite number of times, resulting in a partial, but finite, derivation tree. If all leaves of

the partial derivation tree are axioms, then there is a derivation of $\emptyset \vdash \tau_1 \cong \tau_2$ and, hence, we know: $\tau_1 \cong_D \tau_2$. If not all leaves of the partial derivation tree are axioms, then there is no derivation of $\emptyset \vdash \tau_1 \cong \tau_2$ and, hence, we know: $\tau_1 \not\cong_D \tau_2$.

The decision procedure can be made more efficient by applying rule 1, 2, and 3 only once for the same pair of type variables. If the same pair of type variables occurs in nodes on two different paths of the partial derivation tree, it suffices to continue at one node and stop at the other node. The number of pairs of type variables is quadratic in the length of τ_1 and τ_2 , because the number of type variables in a type is linear in the length of the type. Since syntactic comparison of types (without unfolding) can be done in polynomial time and the number of unfolding operations is at most the number of pairs of type variables, we can conclude that testing for type equivalence is polynomial time decidable.

4.3 Subtyping

In this section, we extend type equivalence to subtyping. More precisely, we define two notions of semantic subtyping and show that they are interchangeable and decidable using a derivation system.

The intuitive idea behind subtyping is the following: τ_1 is a subtype of τ_2 if τ_1 has at least the ‘properties’ of τ_2 . If the properties are preserved by type equivalence, then subtyping is an extension of type equivalence. Similar to type equivalence, trees and extensions can be used to define two notions of semantic subtyping. Structural subtyping is defined as a supertree relation on the trees of the types.

Definition 4.45 Let τ_1 and τ_2 be types. Structural subtyping, where $\tau_1 \preceq_{\text{struct}} \tau_2$ denotes that τ_1 is a structural subtype of τ_2 , is defined by:

$$\tau_1 \preceq_{\text{struct}} \tau_2 \Leftrightarrow \text{struct}(\tau_1) \sqsupseteq \text{struct}(\tau_2).$$

□

Extensional subtyping is defined as a ‘set of subinstances’ relation on the extensions of the types.

Definition 4.46 Let τ_1 and τ_2 be types. Extensional subtyping, where $\tau_1 \preceq_{\text{ext}} \tau_2$ denotes that τ_1 is an extensional subtype of τ_2 , is defined by:

$$\tau_1 \preceq_{\text{ext}} \tau_2 \Leftrightarrow \text{ext}(\tau_1) \preceq_{\text{TERMS}} \text{ext}(\tau_2).$$

□

4.3.1 Derivation system

In this subsection, we introduce a derivation system for subtyping. Again, a derivation is a tree of formulas, where the children formulas imply the parent formula. A formula is of the form $\Gamma \vdash \tau \preceq \sigma$ (where τ and σ are types, \preceq is the subtype relation, and Γ is a context), saying that $\tau \preceq \sigma$ follows from the axioms for basic types and the premises in Γ . A context Γ is a triple $(\Gamma_l, \Gamma_r, \Gamma_p)$, similar to a context in the derivation system for type equivalence.

Definition 4.47 The derivation system for subtyping, denoted by DS , is defined as follows. The axioms of the derivation system are:

1. $\Gamma \vdash B \preceq B$ if $B \in BTypes$
2. $\Gamma \vdash t \preceq s$ if $(t, s) \in \Gamma_p$
3. $\Gamma \vdash t \preceq \mu s.\beta$ if $(t, s) \in \Gamma_p \wedge \mu s.\beta \in \Gamma_r$
4. $\Gamma \vdash \mu t.\alpha \preceq s$ if $(t, s) \in \Gamma_p \wedge \mu t.\alpha \in \Gamma_l$
5. $\Gamma \vdash \mu t.\alpha \preceq \mu s.\beta$ if $(t, s) \in \Gamma_p \wedge \mu t.\alpha \in \Gamma_l \wedge \mu s.\beta \in \Gamma_r$.

The rules of the derivation system are:

1.
$$\frac{\Gamma \vdash \mu t.\alpha \preceq \mu s.\beta}{\Gamma \vdash t \preceq s} \quad \text{if } (t, s) \notin \Gamma_p \wedge \mu t.\alpha \in \Gamma_l \wedge \mu s.\beta \in \Gamma_r$$
2.
$$\frac{\Gamma \vdash \mu t.\alpha \preceq \mu s.\beta}{\Gamma \vdash t \preceq \mu s.\beta} \quad \text{if } (t, s) \notin \Gamma_p \wedge \mu t.\alpha \in \Gamma_l$$
3.
$$\frac{\Gamma \vdash \mu t.\alpha \preceq \mu s.\beta}{\Gamma \vdash \mu t.\alpha \preceq s} \quad \text{if } (t, s) \notin \Gamma_p \wedge \mu s.\beta \in \Gamma_r$$
4.
$$\frac{\Gamma \vdash \tau \preceq \sigma}{\Gamma \vdash \{\tau\} \preceq \{\sigma\}}$$
5.
$$\frac{\Gamma \vdash \tau_1 \preceq \sigma_1, \dots, \Gamma \vdash \tau_n \preceq \sigma_n}{\Gamma \vdash \langle l_1 : \tau_1, \dots, l_n : \tau_n, \dots, l_{n+m} : \tau_{n+m} \rangle \preceq \langle l_1 : \sigma_1, \dots, l_n : \sigma_n \rangle}$$
6.
$$\frac{\Gamma \cup \{(\mu t.\alpha, \mu s.\beta), (t, s)\} \vdash \alpha \preceq \beta}{\Gamma \vdash \mu t.\alpha \preceq \mu s.\beta} \quad \text{if } (t, s) \notin \Gamma_p.$$

□

The set of axioms and rules of DS is the same as the extended set of rules for $<_A$ from [3] and an extension of the well-known subtype rules from [13] with rules for recursive types.

Example 4.48 For convenience, we define a number of abbreviations:

$$\begin{aligned}
\alpha &= \langle a_1 : B, a_2 : t, a_3 : t, a_4 : t \rangle \\
\beta &= \langle a_1 : B, a_2 : s, a_3 : \mu s'. \beta' \rangle \\
\beta' &= \langle a_1 : B, a_2 : s' \rangle \\
\Gamma_1 &= \{(\mu t. \alpha, \mu s. \beta), (t, s)\} \\
\Gamma_2 &= \Gamma_1 \cup \{(\mu t. \alpha, \mu s'. \beta'), (t, s')\}.
\end{aligned}$$

Using derivation system DS , we obtain the following derivation for $\emptyset \vdash \mu t. \alpha \preceq \mu s. \beta$:

$$\begin{array}{c}
\frac{\Gamma_2 \vdash B \preceq B, \Gamma_2 \vdash t \preceq s'}{\Gamma_2 \vdash \alpha \preceq \beta'} \text{ (rule 5)} \\
\frac{\Gamma_2 \vdash \alpha \preceq \beta'}{\Gamma_1 \vdash \mu t. \alpha \preceq \mu s'. \beta'} \text{ (rule 6)} \\
\frac{\Gamma_1 \vdash \mu t. \alpha \preceq \mu s'. \beta'}{\Gamma_1 \vdash B \preceq B, \Gamma_1 \vdash t \preceq s, \Gamma_1 \vdash t \preceq \mu s'. \beta'} \text{ (rule 2)} \\
\frac{\Gamma_1 \vdash B \preceq B, \Gamma_1 \vdash t \preceq s, \Gamma_1 \vdash t \preceq \mu s'. \beta'}{\Gamma_1 \vdash \alpha \preceq \beta} \text{ (rule 5)} \\
\frac{\Gamma_1 \vdash \alpha \preceq \beta}{\emptyset \vdash \mu t. \alpha \preceq \mu s. \beta} \text{ (rule 6)}
\end{array}$$

□

Derivable subtyping of μ -complete types is given by the following definition.

Definition 4.49 Let τ_1 and τ_2 be closed μ -complete types. Derivable subtyping, where $\tau_1 \preceq_D \tau_2$ denotes that τ_1 is a subtype of τ_2 according to derivation system DS , is defined by:

$$\tau_1 \preceq_D \tau_2 \Leftrightarrow \emptyset \vdash_{DS} \tau_1 \preceq \tau_2,$$

where $\Gamma \vdash_{DS} \tau \cong \sigma$ means that there is a derivation in DS with conclusion $\Gamma \vdash \tau \preceq \sigma$. □

In the remainder of this chapter, we will prove Theorems 4.50 through 4.52.

Theorem 4.50 Derivable subtyping is sound and complete w.r.t. structural subtyping. □

Theorem 4.51 Structural and extensional subtyping are logically equivalent. □

Combining Theorems 4.50 and 4.51, we can conclude all notions of type equivalence are logically equivalent. This means that all notions of subtyping are interchangeable. Furthermore, soundness of derivable subtyping w.r.t. extensional subtyping means that if a type is a subtype of another type, then any instance of the first type can be canonically transformed (using a projection function) into an instance of the second type.

Theorem 4.52 Derivable subtyping is decidable. □

Combining Theorems 4.50 through 4.52, we can conclude that all notions of type equivalence are decidable. Furthermore, combining Theorems 4.22 and 4.50, we can deduce that derivable equivalence implies derivable subtyping and that derivable subtyping is antisymmetric w.r.t. derivable equivalence.

Lemma 4.53 Let τ_1 and τ_2 be closed μ -complete types. Then:

$$\tau_1 \cong_D \tau_2 \Leftrightarrow (\tau_1 \preceq_D \tau_2 \wedge \tau_2 \preceq_D \tau_1).$$

Proof.

$$\begin{aligned} \tau_1 \cong_D \tau_2 &\Leftrightarrow \\ \tau_1 &\cong_{\text{struc}} \tau_2 \Leftrightarrow \\ \text{struc}(\tau_1) &= \text{struc}(\tau_2) \Leftrightarrow \\ (\text{struc}(\tau_1) &\sqsupseteq \text{struc}(\tau_2) \wedge \text{struc}(\tau_2) \sqsupseteq \text{struc}(\tau_1)) \Leftrightarrow \\ (\tau_1 &\preceq_{\text{struc}} \tau_2 \wedge \tau_2 \preceq_{\text{struc}} \tau_1) \Leftrightarrow \\ \tau_1 &\preceq_D \tau_2 \wedge \tau_2 \preceq_D \tau_1. \end{aligned}$$

□

For the following chapters, the subtype relation on types is defined as extensional subtyping.

Definition 4.54 Let τ_1 and τ_2 be closed types. The subtype relation, where $\tau_1 \preceq_{\text{TYPE}} \tau_2$ denotes that τ_1 is a subtype of τ_2 , is defined as:

$$\tau_1 \preceq_{\text{TYPE}} \tau_2 \Leftrightarrow \tau_1 \preceq_{\text{ext}} \tau_2.$$

□

4.3.2 Soundness w.r.t structural subtyping

In this subsection, we prove the soundness part of Theorem 4.50. More precisely, for closed μ -complete types τ_1 and τ_2 , we prove:

$$\tau_1 \preceq_D \tau_2 \Rightarrow \tau_1 \preceq_{\text{struc}} \tau_2.$$

The proof is similar to the proof of the soundness part of Theorem 4.50. First, for every formula $\Gamma \vdash \tau \preceq \sigma$ in the derivation of $\emptyset \vdash \tau_1 \preceq \tau_2$, an l-tree and an r-tree are constructed (the l-tree will be proven to be a supertree of the r-tree). The l-tree is constructed in such a way that it is equal to $\text{struc}(\tau, \Gamma_l)$ and the r-tree is constructed in such a way that it is equal to $\text{struc}(\sigma, \Gamma_r)$, except for the free type variables. Following the derivation, constructing the tree for a formula from the trees for its children formulas, we obtain an l-tree that is equal to $\text{struc}(\tau_1, \emptyset)$ and an r-tree that is equal to $\text{struc}(\tau_2, \emptyset)$ (because τ_1 and τ_2 have no free type variables).

The l-tree and the r-tree are the same as the tree in the proof of soundness w.r.t. structural equivalence, except for formulas with record types. For a formula $\Gamma \vdash \tau \preceq \sigma$, where τ and σ are record types, the l-tree has a child for every field in subtype τ and the r-tree has a child for every field in supertype σ .

Definition 4.55 Let τ_1 and τ_2 be closed μ -complete types. Furthermore, let f be an injective function from $bvars(\tau_1) \times bvars(\tau_2)$ to $TypeVar - (bvars(\tau_1) \cup bvars(\tau_2))$ and $\Gamma \vdash \tau \preceq \sigma$ be a formula in the derivation of $\emptyset \vdash \tau_1 \preceq \tau_2$. The l-tree for $\Gamma \vdash \tau \preceq \sigma$, denoted by $tree_l(\tau, \sigma, \Gamma)$ and the r-tree $\Gamma \vdash \tau \preceq \sigma$, denoted by $tree_r(\tau, \sigma, \Gamma)$, are defined as follows:

$$tree_j(B, B, \Gamma) = \begin{array}{c} \textcircled{B} \\ \text{if } B \in BTypes \end{array}$$

$$tree_j(\{\tau_1\}, \{\sigma_1\}, \Gamma) = \begin{array}{c} \textcircled{\{\}} \\ \downarrow \in \\ T \end{array}$$

$$\text{where } T = tree_j(\tau_1, \sigma_1, \Gamma)$$

$$tree_l(\langle l_1 : \tau_1, \dots, l_n : \tau_n, \dots, l_{n+m} : \tau_{n+m} \rangle, \langle l_1 : \sigma_1, \dots, l_n : \sigma_n \rangle, \Gamma) =$$

$$\begin{array}{c} \textcircled{\langle \rangle} \\ \swarrow \quad \searrow \\ l_1 \quad l_{n+m} \\ \downarrow \quad \downarrow \\ T_1 \quad \dots \quad T_{n+m} \end{array}$$

$$\begin{array}{l} \text{where } T_i = tree_l(\tau_i, \sigma_i, \Gamma) \text{ for } i \in \{1, \dots, n\} \\ \text{and } T_i = struc(\tau_i, \Gamma_l) \text{ for } i \in \{n+1, \dots, n+m\} \end{array}$$

$$tree_r(\langle l_1 : \tau_1, \dots, l_n : \tau_n, \dots, l_{n+m} : \tau_{n+m} \rangle, \langle l_1 : \sigma_1, \dots, l_n : \sigma_n \rangle, \Gamma) =$$

$$\begin{array}{c} \textcircled{\langle \rangle} \\ \swarrow \quad \searrow \\ l_1 \quad l_n \\ \downarrow \quad \downarrow \\ T_1 \quad \dots \quad T_n \end{array}$$

$$\text{where } T_i = tree_r(\tau_i, \sigma_i, \Gamma) \text{ for } i \in \{1, \dots, n\}$$

$$tree_j(t, s, \Gamma) = tree_j(t, \mu s.\beta, \Gamma) = tree_j(\mu t.\alpha, s, \Gamma) = tree_j(\mu t.\alpha, \mu s.\beta, \Gamma) =$$

$$\textcircled{f(t,s)}$$

if $(t, s) \in \Gamma_p$

$$\begin{aligned} \text{tree}_j(t, s, \Gamma) &= \text{tree}_j(\mu t.\alpha, \mu s.\beta, \Gamma) \\ \text{if } (t, s) \notin \Gamma_p \wedge \eta_{\Gamma_l}(t) &= \mu t.\alpha \wedge \eta_{\Gamma_r}(s) = \mu s.\beta \end{aligned}$$

$$\begin{aligned} \text{tree}_j(t, \mu s.\beta, \Gamma) &= \text{tree}_j(\mu t.\alpha, \mu s.\beta, \Gamma) \\ \text{if } (t, s) \notin \Gamma_p \wedge \eta_{\Gamma_l}(t) &= \mu t.\alpha \end{aligned}$$

$$\begin{aligned} \text{tree}_j(\mu t.\alpha, s, \Gamma) &= \text{tree}_j(\mu t.\alpha, \mu s.\beta, \Gamma) \\ \text{if } (t, s) \notin \Gamma_p \wedge \eta_{\Gamma_r}(s) &= \mu s.\beta \end{aligned}$$

$$\begin{aligned} \text{tree}_j(\mu t.\alpha, \mu s.\beta, \Gamma) &= \text{tree}_j(\alpha, \beta, \Gamma \cup \{(\mu t.\alpha, \mu s.\beta)\}) \\ \text{if } (t, s) \notin \Gamma_p, \end{aligned}$$

where $j \in \{l, r\}$. \square

Finally, we prove soundness in two steps. First, we prove Claim 5:

$$\emptyset \vdash_{DS} \tau_1 \preceq \tau_2 \Rightarrow \text{tree}_l(\tau_1, \tau_2, \emptyset) \sqsupseteq \text{tree}_r(\tau_1, \tau_2, \emptyset),$$

and second, we prove Claim 6:

$$\begin{aligned} \emptyset \vdash_{DS} \tau_1 \preceq \tau_2 \Rightarrow & (\text{tree}_l(\tau_1, \tau_2, \emptyset) = \text{struc}(\tau_1, \emptyset) \wedge \\ & \text{tree}_r(\tau_1, \tau_2, \emptyset) = \text{struc}(\tau_2, \emptyset)). \end{aligned}$$

From these claims it follows that derivable subtyping is sound w.r.t. structural subtyping:

$$\emptyset \vdash_{DS} \tau_1 \preceq \tau_2 \Rightarrow \text{struc}(\tau_1) \sqsupseteq \text{struc}(\tau_2).$$

4.3.2.1 Proof of Claim 5

We prove the following, stronger, version of Claim 5. For every formula $\Gamma \vdash \tau \preceq \sigma$ in the derivation tree of $\emptyset \vdash \tau_1 \preceq \tau_2$:

$$\text{tree}_l(\tau, \sigma, \Gamma) \sqsupseteq \text{tree}_r(\tau, \sigma, \Gamma).$$

The proof is an induction argument on the distance of a formula in the derivation tree to its remotest descendant. Base step: the formula is an axiom.

Axiom 1: $\Gamma \vdash B \preceq B$. Then, obviously, $\text{tree}_l(B, B, \Gamma) \sqsupseteq \text{tree}_r(B, B, \Gamma)$.

Axiom 2: $\Gamma \vdash t \preceq s$. Then (t, s) must be an element of Γ . Hence, $\text{tree}_l(t, s, \Gamma) \sqsupseteq \text{tree}_r(t, s, \Gamma)$.

Axiom 3, 4, or 5: $\Gamma \vdash t \preceq \mu s.\beta$, $\Gamma \vdash \mu t.\alpha \preceq s$, or $\Gamma \vdash \mu t.\alpha \preceq \mu s.\beta$. Similar to the

previous case.

Induction step: the formula is the result of applying a rule to a number of formulas which are closer to their remotest descendant.

Rule 1: $\Gamma \vdash t \preceq s$. Then there must be types $\eta_{\Gamma_l}(t) = \mu t.\alpha$ and $\eta_{\Gamma_r}(s) = \mu s.\beta$, such that: $\Gamma \vdash \mu t.\alpha \preceq \mu s.\beta$. Using the induction hypothesis, we can conclude: $tree_l(\mu t.\alpha, \mu s.\beta, \Gamma) \supseteq tree_r(\mu t.\alpha, \mu s.\beta, \Gamma)$. Since (t, s) is not an element of Γ , it follows that:

$$tree_l(t, s, \Gamma) = tree_l(\mu t.\alpha, \mu s.\beta, \Gamma) \supseteq tree_r(\mu t.\alpha, \mu s.\beta, \Gamma) = tree_r(t, s, \Gamma).$$

Rule 2 and 3: $\Gamma \vdash t \preceq \mu s.\beta$ or $\Gamma \vdash \mu t.\alpha \preceq s$. Similar to the previous case.

Rule 4: $\Gamma \vdash \{\tau\} \preceq \{\sigma\}$. Apply the induction hypothesis to $\Gamma \vdash \tau \preceq \sigma$.

Rule 5: $\Gamma \vdash \langle l_1 : \tau_1, \dots, l_n : \tau_n, \dots, l_{n+m} : \tau_{n+m} \rangle \preceq \langle l_1 : \sigma_1, \dots, l_n : \sigma_n \rangle$. Apply the induction hypothesis to $\vdash \tau_i \preceq \sigma_i$.

Rule 6: $\Gamma \vdash \mu t.\alpha \preceq \mu s.\beta$. Let Γ' be $\Gamma \cup \{(\mu t.\alpha, \mu s.\beta)\}$ and Δ be $\Gamma' \cup \{(t, s)\}$. From $\Delta \vdash \alpha \preceq \beta$ and the induction hypothesis, it follows that:

$$a) tree_l(\alpha, \beta, \Delta) \supseteq tree_r(\alpha, \beta, \Delta).$$

The proof is based on a) and the fact that $tree_j(\mu t.\alpha, \mu s.\beta, \Gamma)$ is an infinite repetition of $tree_j(\alpha, \beta, \Delta)$. To formalise the infinite repetition of $tree_j(\alpha, \beta, \Delta)$, we define $tree_{j,i}(\alpha, \beta, \Delta)$ for $j \in \{l, r\}$ and $i \in \mathbb{N}$:

$$\begin{aligned} tree_{j,1}(\alpha, \beta, \Delta) &= tree_j(\alpha, \beta, \Delta), \\ tree_{j,i+1}(\alpha, \beta, \Delta) &= tree_j(\alpha, \beta, \Delta)[f(t, s) \setminus tree_{j,i}(\alpha, \beta, \Delta)]. \end{aligned}$$

Then, for every $i \in \mathbb{N}$, we have the following two properties (which will be proven):

$$\begin{aligned} b) tree_{j,i}(\alpha, \beta, \Delta)[f(t, s) \setminus tree_j(\alpha, \beta, \Gamma')] &= tree_j(\alpha, \beta, \Gamma') \\ c) tree_{l,i}(\alpha, \beta, \Delta) &\supseteq tree_{r,i}(\alpha, \beta, \Delta). \end{aligned}$$

Property b) states that $tree_{j,i}(\alpha, \beta, \Delta)$ is equal to $tree_j(\alpha, \beta, \Gamma')$ from the root to at least depth i . Using property c), we can deduce that $tree_l(\alpha, \beta, \Gamma')$ is a supertree of $tree_r(\alpha, \beta, \Gamma')$. Hence:

$$tree_l(\mu t.\alpha, \mu s.\beta, \Gamma) = tree_l(\alpha, \beta, \Gamma') \supseteq tree_r(\alpha, \beta, \Gamma') = tree_r(\mu t.\alpha, \mu s.\beta, \Gamma).$$

The proof of b) is the same as the proof of b) in the proof of Claim 1. The proof of c) is an induction argument on i , where the base step follows from a) and the induction step is:

$$\begin{aligned} tree_{l,i+1}(\alpha, \beta, \Delta) &= tree_l(\alpha, \beta, \Delta)[f(t, s) \setminus tree_{l,i}(\alpha, \beta, \Delta)] \supseteq \\ &tree_r(\alpha, \beta, \Delta)[f(t, s) \setminus tree_{l,i}(\alpha, \beta, \Delta)] \supseteq \\ &tree_r(\alpha, \beta, \Delta)[f(t, s) \setminus tree_{r,i}(\alpha, \beta, \Delta)] = \\ &tree_{r,i+1}(\alpha, \beta, \Delta). \end{aligned}$$

4.3.2.2 Proof of Claim 6

In the same way as Claim 1 was proven, we can prove the following, stronger, version of Claim 6. For every formula $\Gamma \vdash \tau \preceq \sigma$ in the derivation tree of $\emptyset \vdash \tau_1 \preceq \tau_2$:

- 1) $tree_l(\tau, \sigma, \Gamma)[f(t', s') \setminus struc(\mu t'.\alpha', \Gamma_l) \mid (t', s') \in \Gamma_p \wedge \eta_{\Gamma_l}(t') = \mu t'.\alpha'] = struc(\tau, \Gamma_l)$
- 2) $tree_r(\tau, \sigma, \Gamma)[f(t', s') \setminus struc(\mu s'.\beta', \Gamma_r) \mid (t', s') \in \Gamma_p \wedge \eta_{\Gamma_r}(s') = \mu s'.\beta'] = struc(\sigma, \Gamma_r).$

4.3.3 Completeness w.r.t. structural subtyping

In this subsection, we prove the completeness part of Theorem 4.50. More precisely, for closed μ -complete types τ_1 and τ_2 , we prove:

$$\tau_1 \preceq_{struct} \tau_2 \Rightarrow \tau_1 \preceq_D \tau_2.$$

The proof is similar to the proof of the completeness part of Theorem 4.22. First, a structural subtyping tree for τ_1 and τ_2 is constructed (the structural subtyping tree will be proven isomorphic to the derivation tree with conclusion $\emptyset \vdash \tau_1 \preceq \tau_2$). The tree is constructed in such a way that every node is labeled by a tuple of the form (τ, σ, Γ) , where τ and σ are obtained by using the structure of τ_1 and τ_2 (and, if necessary, by unfolding of types), such that $struc(\tau, \Gamma_l) \supseteq struc(\sigma, \Gamma_r)$. For example, the root is labeled $(\tau_1, \tau_2, \emptyset)$.

For the definition of structural subtyping trees, we need the following Lemma.

Lemma 4.56 Let τ and σ be μ -complete types, and Γ be a context, such that $struc(\tau, \Gamma_l) \supseteq struc(\sigma, \Gamma_r)$ and $struc(\tau, \Gamma_l)$ contains no type variables. Then:

1. if $\tau = B$, then $\sigma = B$
2. if $\tau = \{\tau_1\}$, then $\sigma = \{\sigma_1\}$ and $struc(\tau_1, \Gamma_l) \supseteq struc(\sigma_1, \Gamma_r)$
3. if $\tau = \langle l_1 : \tau_1, \dots, l_m : \tau_m \rangle$, then $\sigma = \langle l_1 : \sigma_1, \dots, l_n : \sigma_n \rangle$ and $m \geq n$ and $struc(\tau_i, \Gamma_l) \supseteq struc(\sigma_i, \Gamma_r)$
4. if $\tau = t$, then either
 - a. $\sigma = s$ and $\eta_{\Gamma_l}(t) = \mu t.\alpha$ and $\eta_{\Gamma_r}(s) = \mu s.\beta$ and $struc(\alpha, \Gamma_l) \supseteq struc(\beta, \Gamma_r)$
 - b. $\sigma = \mu s.\beta$ and $\eta_{\Gamma_l}(t) = \mu t.\alpha$ and $struc(\alpha, \Gamma_l) \supseteq struc(\beta, \Gamma_r \cup \{\sigma\})$
5. if $\tau = \mu t.\alpha$, then either
 - a. $\sigma = s$ and $\eta_{\Gamma_r}(s) = \mu s.\beta$ and $struc(\alpha, \Gamma_l \cup \{\tau\}) \supseteq struc(\beta, \Gamma_r)$
 - b. $\sigma = \mu s.\beta$ and $struc(\alpha, \Gamma_l \cup \{\tau\}) \supseteq struc(\beta, \Gamma_r \cup \{\sigma\})$.

Proof. The lemma follows from Definition 4.3. \square

The children of a node in a structural subtyping tree are the same as the children in a structural equivalence tree, except for nodes with record types. For a node

(τ, σ, Γ) , where τ and σ are record types, there is one child for every field in supertype σ , but no child for the additional fields in subtype τ .

Definition 4.57 Let τ and σ be μ -complete types, and Γ be a context, such that $struc(\tau, \Gamma_l) \supseteq struc(\sigma, \Gamma_r)$ and $struc(\tau, \Gamma_l)$ contains no type variables. Furthermore, let x be (τ, σ, Γ) . According to Lemma 4.56, there are 5 cases for x , of which the last two cases each have two subcases. For the definition of the children of x , we divide each subcase into two new subcases (one for $(t, s) \in \Gamma_p$ and one for $(t, s) \notin \Gamma_p$), obtaining 11 cases for x . The set of children of x , denoted by $stchildren(x)$ is defined as follows:

1. if $x = (B, B, \Gamma)$, then $stchildren(x) = \emptyset$
2. if $x = (t, s, \Gamma)$ and $(t, s) \in \Gamma_p$, then $stchildren(x) = \emptyset$
3. if $x = (t, \mu s.\beta, \Gamma)$ and $(t, s) \in \Gamma_p$, then $stchildren(x) = \emptyset$
4. if $x = (\mu t.\alpha, s, \Gamma)$ and $(t, s) \in \Gamma_p$, then $stchildren(x) = \emptyset$
5. if $x = (\mu t.\alpha, \mu s.\beta, \Gamma)$ and $(t, s) \in \Gamma_p$, then $stchildren(x) = \emptyset$
6. if $x = (\{\tau\}, \{\sigma\}, \Gamma)$, then $stchildren(x) = \{(\tau, \sigma, \Gamma)\}$
7. if $x = (\langle l_1 : \tau_1, \dots, l_{n+m} : \tau_{n+m} \rangle, \langle l_1 : \sigma_1, \dots, l_n : \sigma_n \rangle, \Gamma)$, then $stchildren(x) = \{(\tau_1, \sigma_1, \Gamma), \dots, (\tau_n, \sigma_n, \Gamma)\}$
8. if $x = (t, s, \Gamma)$ and $(t, s) \notin \Gamma_p \wedge \mu t.\alpha \in \Gamma_l \wedge \mu s.\beta \in \Gamma_r$, then $stchildren(x) = \{(\mu t.\alpha, \mu s.\beta, \Gamma)\}$
9. if $x = (t, \mu s.\beta, \Gamma)$ and $(t, s) \notin \Gamma_p \wedge \mu t.\alpha \in \Gamma_l$, then $stchildren(x) = \{(\mu t.\alpha, \mu s.\beta, \Gamma)\}$
10. if $x = (\mu t.\alpha, s, \Gamma)$ and $(t, s) \notin \Gamma_p \wedge \mu s.\beta \in \Gamma_r$, then $stchildren(x) = \{(\mu t.\alpha, \mu s.\beta, \Gamma)\}$
11. if $x = (\mu t.\alpha, \mu s.\beta, \Gamma)$ and $(t, s) \notin \Gamma_p$, then $stchildren(x) = \{(\alpha, \beta, \Gamma \cup \{(\mu t.\alpha, \mu s.\beta), (t, s)\})\}$.

□

The following lemma states that, if a tuple satisfies the precondition of Definition 4.57, then so do its children.

Lemma 4.58 Let $x = (\tau, \sigma, \Gamma)$ be a tuple, such that τ and σ are μ -complete types, Γ is a context, $struc(\tau, \Gamma_l) \supseteq struc(\sigma, \Gamma_r)$, and $struc(\tau, \Gamma_l)$ contains no type variables. Then every element of $stchildren(x)$ is a tuple $x' = (\tau', \sigma', \Gamma')$, such that $struc(\tau', \Gamma'_l)$ is a supertree of $struc(\sigma', \Gamma'_r)$ and $struc(\tau', \Gamma'_l)$ contains no type variables.

Proof. The lemma follows from Definition 4.57 and Lemma 4.56. □

Using Lemma 4.58, we can finally define structural subtyping trees.

Definition 4.59 Let τ_1 and τ_2 be closed μ -complete types, such that $struc(\tau_1, \emptyset) \supseteq struc(\tau_2, \emptyset)$. The structural subtyping tree for τ_1 and τ_2 is defined as $sttree((\tau_1, \tau_2, \emptyset))$, where $sttree((\tau, \sigma, \Gamma))$ is defined as follows:

if $x = (\tau, \sigma, \Gamma)$ and $stchildren(x) = \emptyset$,
 then $sttree(x)$ has only one node, labeled (τ, σ, Γ)
 if $x = (\tau, \sigma, \Gamma)$ and $stchildren(x) \neq \emptyset$,
 then $sttree(x)$ consists of a root, labeled (τ, σ, Γ) , and,
 for every $y \in stchildren(x)$, a subtree $sttree(y)$ and an arrow
 from the root to subtree $sttree(y)$.

□

Lemma 4.60 The structural subtyping tree for τ_1 and τ_2 is finite.

Proof. The proof is the same as the proof of Lemma 4.34. □

Finally, in the same way as Claim 2 was proven, we can prove the following claim:

for every node labeled (τ, σ, Γ) in $sttree(\tau_1, \tau_2, \emptyset)$: $\Gamma \vdash_{DS} \tau \preceq \sigma$.

From the definition of $sttree$ and the claim it follows that derivable subtyping is complete w.r.t. structural subtyping:

$$struc(\tau_1) \sqsupseteq struc(\tau_2) \Rightarrow \emptyset \vdash_{DS} \tau_1 \preceq \tau_2.$$

4.3.4 Equivalence of structural and extensional subtyping

In this subsection, we prove Theorem 4.51. More precisely, for closed μ -complete types τ_1 and τ_2 , we prove:

$$\tau_1 \preceq_{struc} \tau_2 \Leftrightarrow \tau_1 \preceq_{ext} \tau_2.$$

The proof is similar to the proof of Theorem 4.23. First, we define the projection of a term on (the tree of) a type. The projection is the same as the projection in the proof of Theorem 4.23, except for records.

Definition 4.61 Let τ_1 and τ_2 be types, such that $struc(\tau_1) \sqsupseteq struc(\tau_2)$. Furthermore, let g be an injective function from $bvars(\tau_1) \times TypeVar$ to $Var - fvars(\tau_1)$ and e be a term, such that $\forall e' \in subs(e)[inst(estruc(e'), estruc(\tau_1))]$. The projection of e on $estruc(\tau_2)$ is defined as $proj(e, estruc(\tau_2), \emptyset)$, where $proj(e', U, V)$ is defined as follows:

$$\begin{aligned} proj(x, node(t), V) &= x \text{ if } x \in Var \\ proj(b, node(B), V) &= b \text{ if } b \in Cons \\ proj(\emptyset, tree(\{T\}), V) &= \emptyset \\ proj(\{e_1, \dots, e_n\}, tree(\{T\}), V) &= \{proj(e_1, T, V), \dots, proj(e_n, T, V)\} \\ proj(x, tree(t. < l_1 : T_1, \dots, l_n : T_n >), V) &= g(x, t) \\ &\text{ if } g(x, t) \in V \\ proj(x, tree(t. < l_1 : T_1, \dots, l_n : T_n >), V) &= \\ &proj(\eta_V(x), tree(t. < l_1 : T_1, \dots, l_n : T_n >), V) \end{aligned}$$

$$\begin{aligned}
& \text{if } g(x, t) \notin V \\
& \text{proj}(\mu x. \langle l_1 = e_1, \dots, l_m = e_m \rangle, \text{tree}(t. \langle l_1 : T_1, \dots, l_n : T_n \rangle), V) = g(x, t) \\
& \text{if } g(x, t) \in V \\
& \text{proj}(\mu x. \langle l_1 = e_1, \dots, l_m = e_m \rangle, \text{tree}(t. \langle l_1 : T_1, \dots, l_n : T_n \rangle), V) = \\
& \quad \mu g(x, t). \langle l_1 : \text{proj}(e_1, T_1, V \cup \{g(x, t), \mu x. \langle l_1 = e_1, \dots, l_m = e_m \rangle\}), \dots, \\
& \quad \quad l_n : \text{proj}(e_n, T_n, V \cup \{g(x, t), \mu x. \langle l_1 = e_1, \dots, l_m = e_m \rangle\}) \rangle \\
& \text{if } g(x, t) \notin V,
\end{aligned}$$

where *node* and *tree* are as defined in Definition 4.43. \square

Finally, in the same way as Claim 3 was proven, we can prove the following claim:

$$\text{struc}(\tau_1) \supseteq \text{struc}(\tau_2) \Leftrightarrow \text{struc_ext}(\tau_1) \supseteq \text{struc_ext}(\tau_2).$$

Using this claim and Claim 4, we can conclude that structural and extensional subtyping are logically equivalent:

$$\begin{aligned}
& \text{struc}(\tau_1) \supseteq \text{struc}(\tau_2) \Leftrightarrow \\
& \text{struc_ext}(\tau_1) \supseteq \text{struc_ext}(\tau_2) \Leftrightarrow \\
& \{\text{struc}(e) \mid e \in \text{ext}(\tau_1)\} \supseteq \{\text{struc}(e) \mid e \in \text{ext}(\tau_2)\} \Leftrightarrow \\
& \text{ext}(\tau_1) \preceq_{\text{TERMS}} \text{ext}(\tau_2).
\end{aligned}$$

4.3.5 Decidability of derivable subtyping

The proof of Theorem 4.52 is similar to the proof of Theorem 4.25. In fact, in the same way as it was proven that there is a polynomial time decision procedure for derivable equivalence, we can prove that there is a polynomial time decision procedure for derivable subtyping.

Chapter 5

Properties of functional forms

In this chapter, we focus on functions to obtain a formal basis for the comparison of methods. We consider functions of which the first domain and codomain are types as defined in the previous chapter and the other domains are basic types. We introduce an equality relation on functions, similar to extensional equality, and weaken it. Furthermore, we define a subfunction relation using generalisations of functions and the equality relation and weaken it. We show that the equality relation and the subfunction relation are polynomial time decidable. These results are important for the comparison of methods.

5.1 Syntax and semantics

In this section, we define functional forms syntactically and semantically. The syntax of functional forms is given by the following definition.

Definition 5.1 Let τ and σ be closed types as defined in the previous chapter. The set of functional forms from τ to σ , denoted by $functions(\tau, \sigma)$, is defined as follows:

$$functions(\tau, \sigma) = \{\lambda obj:\tau \lambda \vec{p}:\vec{P}. e \mid k \in \mathbb{N} \wedge \vec{p} \in \mathcal{L}^k \wedge \vec{P} \in BTypes^k \wedge e \in expr(\tau, \sigma, \emptyset)\}$$

where $obj \notin \mathcal{L}$ is a special label and the set of expressions is given by:

$$\begin{aligned} expr(\tau, t, V) &= Var_t \\ &\text{if } t \in TypeVar \\ expr(\tau, B, V) &= bexpr(\tau, B, V) \cup Cons_B \cup \end{aligned}$$

$$\begin{aligned}
& \{e_1 \theta e_2 \mid e_1 \in \text{expr}(\tau, B, V) \wedge \theta \in \text{ops}_B \wedge e_2 \in \text{expr}(\tau, B, V)\} \\
& \text{if } B \in BTypes \\
& \text{expr}(\tau, \{v\}, V) = \text{bexpr}(\tau, \{v\}, V) \cup \\
& \quad \{e \cup \{e_1, \dots, e_n\} \mid e \in \text{expr}(\tau, \{v\}, V) \wedge \forall i \in \{1, \dots, n\} [e_i \in \text{expr}(\tau, v, V)]\} \\
& \text{expr}(\tau, \langle l_1 : v_1, \dots, l_n : v_n \rangle, V) = \text{bexpr}(\tau, \langle l_1 : v_1, \dots, l_n : v_n \rangle, V) \cup \\
& \quad \{\langle l_1 = e_1, \dots, l_n = e_n \rangle \mid \forall j \in \{1, \dots, n\} [e_j \in \text{expr}(\tau, v_j, V)]\} \\
& \text{expr}(\tau, \mu t. \alpha, V) = \text{bexpr}(\tau, \mu t. \alpha, V) \cup \\
& \quad \{\mu x. (e_0[x_1 \setminus e_1, \dots, x_n \setminus e_n]) \mid x \in \text{Var}_t \wedge \\
& \quad e_0 \in \text{expr}(\tau, \alpha, V \cup \{\mu t. \alpha\}) \wedge \\
& \quad FV(e_0) \cap \text{Var}_t = \{x_1, \dots, x_n\} \wedge \\
& \quad \forall j \in \{1, \dots, n\} [e_j \in \text{Var}_t \cup \text{expr}(\tau, \mu t. \alpha, V) \wedge x \notin BV(e_j)]\},
\end{aligned}$$

where the set of basic expressions is given by:

$$\begin{aligned}
\text{bexpr}(\tau, v, V) = & \{obj.l_1 \dots l_n \mid \text{estruc}(\tau.l_1 \dots l_n) = \text{estruc}(v, V)\} \\
& \cup \{p_i \mid \text{estruc}(P_i) = \text{estruc}(v, V)\}
\end{aligned}$$

and the set of operators is given by:

$$\begin{aligned}
\text{ops}_{\text{oid}} &= \emptyset \\
\text{ops}_{\text{integer}} &= \{+, -, \times\} \\
\text{ops}_{\text{rational}} &= \{+, -, \times, \div\} \\
\text{ops}_{\text{string}} &= \{+\}.
\end{aligned}$$

The set of all functional forms, denoted by *Functions*, is defined as:

$$\text{Functions} = \bigcup \{\text{functions}(\tau, \sigma) \mid \tau \in \text{Types} \wedge \sigma \in \text{Types}\}.$$

□

The semantics of a functional form from τ to σ can be defined as a set-theoretic function, where the first domain of the function is given by the semantics of τ and the codomain of the function is given by the semantics of σ .

Definition 5.2 Let the semantics of a type v , denoted by $\llbracket v \rrbracket$, be given by:

$$\llbracket v \rrbracket = \{\text{ext}(v') \mid v' \preceq_{\text{TYPE}} v\}.$$

Furthermore, let τ and σ be types and $F = \lambda obj : \tau \lambda p_1 : \tau_1 \dots p_k : \tau_k. f$ be a functional form in $\text{functions}(\tau, \sigma)$. The semantics of F are given by function F_{\square} from $\llbracket \tau \rrbracket \times \llbracket \tau_1 \rrbracket \times \dots \times \llbracket \tau_k \rrbracket$ to $\llbracket \sigma \rrbracket$, such that:

$$\begin{aligned}
& \forall e_0 \in \llbracket \tau \rrbracket \forall e_1 \in \llbracket \tau_1 \rrbracket \dots \forall e_k \in \llbracket \tau_k \rrbracket \\
& [F_{\square}(e_0, e_1, \dots, e_k) = \mathcal{I}(F(e_0)(e_1) \dots (e_k))],
\end{aligned}$$

where

$$F(e_0)(e_1) \dots (e_k) = f[obj \setminus e_0, p_1 \setminus e_1, \dots, p_k \setminus e_k]$$

and

$$\begin{aligned}
\mathcal{I}(x) &= x \text{ if } x \in \text{Var} \\
\mathcal{I}(b) &= b \text{ if } b \in \text{Cons} \\
\mathcal{I}(\delta(\mu x.e)) &= x \\
\mathcal{I}(e.l_1 \cdots l_n) &= \mathcal{I}(\mathcal{I}(e.l_1 \cdots l_{n-1}).l_n) \\
&\quad \text{if } n > 1 \\
\mathcal{I}(< l_1 = e_1, \dots, l_n = e_n > .l_i) &= e_i \\
\mathcal{I}((\mu x.e).l) &= \mathcal{I}(\text{cut}(e[x \setminus \mu x.e], \emptyset).l) \\
\mathcal{I}(e_1 \theta e_2) &= \mathcal{E}(\theta, \mathcal{I}(e_1), \mathcal{I}(e_2)) \\
\mathcal{I}(\{e_1, \dots, e_n\}) &= \{\mathcal{I}(e_1), \dots, \mathcal{I}(e_n)\} \\
\mathcal{I}(< l_1 = e_1, \dots, l_n = e_n >) &= < l_1 = \mathcal{I}(e_1), \dots, l_n = \mathcal{I}(e_n) > \\
\mathcal{I}(\mu x.e) &= \mu x.\mathcal{I}(e)
\end{aligned}$$

and

$$\begin{aligned}
\text{cut}(x', V) &= x' && \text{if } x' \in \text{Var} \\
\text{cut}(b, V) &= b && \text{if } b \in \text{Cons} \\
\text{cut}(\{e_1, \dots, e_n\}, V) &= \{\text{cut}(e_1, V), \dots, \text{cut}(e_n, V)\} \\
\text{cut}(< l_1 = e_1, \dots, l_n : e_n >, V) &= < l_1 = \text{cut}(e_1, V), \dots, l_n = \text{cut}(e_n, V) > \\
\text{cut}(\mu x'.e', V) &= \mu x'.(\text{cut}(e', V \cup \{x'\})) && \text{if } x' \notin V \\
\text{cut}(\mu x'.e', V) &= x' && \text{if } x' \in V
\end{aligned}$$

and

$$\begin{aligned}
\mathcal{E}(+, g_1, g_2) &\text{ is the sum of } g_1 \text{ and } g_2 \\
&\quad \text{if } g_1, g_2 \in \text{Cons}_{\text{integer}} \text{ or } g_1, g_2 \in \text{Cons}_{\text{rational}} \\
\mathcal{E}(+, g_1, g_2) &\text{ is the concatenation of } g_1 \text{ and } g_2 \\
&\quad \text{if } g_1, g_2 \in \text{Cons}_{\text{string}} \\
\mathcal{E}(-, g_1, g_2) &\text{ is the difference of } g_1 \text{ and } g_2 \\
&\quad \text{if } g_1, g_2 \in \text{Cons}_{\text{integer}} \text{ or } g_1, g_2 \in \text{Cons}_{\text{rational}} \\
\mathcal{E}(\times, g_1, g_2) &\text{ is the product of } g_1 \text{ and } g_2 \\
&\quad \text{if } g_1, g_2 \in \text{Cons}_{\text{integer}} \text{ or } g_1, g_2 \in \text{Cons}_{\text{rational}} \\
\mathcal{E}(\div, g_1, g_2) &\text{ is the quotient of } g_1 \text{ and } g_2 \\
&\quad \text{if } g_1, g_2 \in \text{Cons}_{\text{rational}} \\
\mathcal{E}(\theta, g_1, g_2) &= \perp \text{ otherwise.}
\end{aligned}$$

□

Equality of functional forms is defined as extensional equality of their semantic counterparts:

Definition 5.3 Let τ and σ be types. Let $F = \lambda \text{obj}:\tau \lambda p_1:\tau_1 \cdots \lambda p_k:\tau_k.f$ and $G = \lambda \text{obj}:\tau \lambda q_1:v_1 \cdots \lambda q_m:v_m.g$ be functions in $\text{functions}(\tau, \sigma)$. Equality of F and G , denoted by $F =_{FUNC} G$, is defined by:

$$F =_{FUNC} G \Leftrightarrow \forall e_0 \in [\tau] \forall e_1 \in [\tau_1] \cdots \forall e_k \in [\tau_k] [F_{\square}(\vec{e}) =_D G_{\square}(\vec{e})],$$

where $\vec{e} = (e_0, e_1, \dots, e_k)$. Note that equality requires that $k = m$ and $\forall j \in \{1, \dots, n\} [\tau_j = v_j]$. \square

Equality of functional forms can be weakened by permuting parameters.

Definition 5.4 Let τ and σ be types. Let $F = \lambda obj:\tau \lambda P.f$ and $G = \lambda obj:\tau \lambda Q.g$ be functions in $functions(\tau, \sigma)$. Then F is weakly equal to G , denoted by $F =_{FUNC}^w G$, if and only if there exists a permutation \tilde{Q} of Q , such that:

$$F =_{FUNC} \lambda obj:\tau \lambda \tilde{Q}.g.$$

\square

To define a subfunction relation on functional forms, we have to introduce the generalisation of a functional form.

Definition 5.5 Let τ_1 and τ_2 be types, such that $\tau_1 \preceq_{TYPE} \tau_2$. The generalisation of a functional form from τ_1 to τ_2 to a functional form from τ_2 to τ_2 is defined as follows:

$$gen(\lambda obj:\tau_1 \lambda P. e, \tau_2) = \lambda obj:\tau_2 \lambda \tilde{P}. (proj(e, estruc(\tau_2), obj, \emptyset)),$$

where \tilde{P} is obtained from P by removing parameters that do not occur in $proj(e, struc(\tau_2), obj, \emptyset)$ and:

$$\begin{aligned} &proj(x, tree(t. < l_1 : T_1, \dots, l_n : T_n >), \bar{l}, V) = g(x, t) \\ &\quad \text{if } x \in Var \text{ and } (x, t) \in V \\ &proj(x, tree(t. < l_1 : T_1, \dots, l_n : T_n >), \bar{l}, V) = \perp \\ &\quad \text{if } x \in Var \text{ and } (x, t) \notin V \\ &proj(p, T, \bar{l}, V) = p \\ &\quad \text{if } p \text{ is a label occurring in } P \\ &proj(obj.l_1 \dots l_n, T, \bar{l}, V) = obj.l_1 \dots l_n \\ &\quad \text{if } \tau_2.l_1 \dots l_n \text{ exists and } T \text{ is a subtree of } estruc(\tau_2.l_1 \dots l_n) \\ &proj(obj.l_1 \dots l_n, T, \bar{l}, V) = \perp \\ &\quad \text{if } \tau_2.l_1 \dots l_n \text{ does not exist or } T \text{ is not a subtree of } estruc(\tau_2.l_1 \dots l_n) \\ &proj(b, T, \bar{l}, V) = b \\ &\quad \text{if } b \in Cons \\ &proj(g_1 \theta g_2, T, \bar{l}, V) = proj(g_1, T, \bar{l}, V) \theta proj(g_2, T, \bar{l}, V) \\ &\quad \text{if } \theta \in \{+, -, \times, \div\} \\ &proj(g \cup \{g_1, \dots, g_n\}, tree(\{T\}), \bar{l}, V) = \\ &\quad proj(g, tree(\{T\}), \bar{l}, V) \cup \\ &\quad \{proj(g_i, T, \bar{l}, V) \mid i \in \{1, \dots, n\} \wedge proj(g_i, T, \bar{l}, V) \neq \perp\} \\ &proj(< l_1 = g_1, \dots, l_{n+m} = g_{n+m} >, tree(< >. < l_1 : T_1, \dots, l_n : T_n >), \bar{l}, V) = \\ &\quad < l_1 = cond(proj(g_1, T_1, \bar{l}.l_1, V), \bar{l}.l_1), \dots, \end{aligned}$$

$$\begin{aligned}
& l_n = \text{cond}(\text{proj}(g_n, T_n, \bar{l}.l_n, V), \bar{l}.l_n) > \\
& \text{proj}(\mu x.e, \text{tree}(t. < l_1 : T_1, \dots, l_n : T_n >), \bar{l}, V) = g(x, t) \\
& \quad \text{if } (x, t) \in V \\
& \text{proj}(\mu x.e, \text{tree}(t. < l_1 : T_1, \dots, l_n : T_n >), \bar{l}, V) = \\
& \quad \mu g(x, t). \text{proj}(e, \text{tree}(t. < l_1 : T_1, \dots, l_n : T_n >), \bar{l}, V \cup \{(x, t)\}) \\
& \quad \text{if } (x, t) \notin V,
\end{aligned}$$

where *tree* is as defined in Definition 4.43, *g* is an injective function from $\text{Var} \times \text{TypeVar}$ to Var , such that $g(x, t) \in \text{Var}_t$, $\text{cond}(e_1, e_2) = e_1$ if $e_1 \neq \perp$, and $\text{cond}(e_1, e_2) = e_2$ if $e_1 = \perp$. \square

The subfunction relation on functional forms is defined as extensional equality of their generalisations.

Definition 5.6 Let τ_1 and τ_2 be types, such that $\tau_1 \preceq_{\text{TYPE}} \tau_2$. Let F_1 be a functional form in $\text{functions}(\tau_1, \tau_1)$ and F_2 be a functional form in $\text{functions}(\tau_2, \tau_2)$. Then F_1 is a subfunction of F_2 , denoted by $F_1 \leq_{\text{FUNC}} F_2$, if and only if:

$$\text{gen}(F_1, \tau_2) =_{\text{FUNC}} F_2.$$

\square

The subfunction relation can be weakened by permuting parameters.

Definition 5.7 Let τ_1 and τ_2 be types, such that $\tau_1 \preceq_{\text{TYPE}} \tau_2$. Let F_1 be a functional form in $\text{functions}(\tau_1, \tau_1)$ and F_2 be a functional form in $\text{functions}(\tau_2, \tau_2)$. Then F_1 is a weak subfunction of F_2 , denoted by $F_1 \leq_{\text{FUNC}}^w F_2$, if and only if:

$$\text{gen}(F_1, \tau_2) =_{\text{FUNC}}^w F_2.$$

\square

Lemma 5.8 Let τ_1 and τ_2 be types, such that $\tau_1 \preceq_{\text{TYPE}} \tau_2$. Let F_1 be a functional form in $\text{functions}(\tau_1, \tau_1)$ and $F_2 = \lambda \text{obj}:\tau_2 \lambda P_2.f_2$ be a functional form in $\text{functions}(\tau_2, \tau_2)$. Then $F_1 \leq_{\text{FUNC}}^w F_2$ if and only if there is a permutation \tilde{P}_2 of P_2 , such that:

$$F_1 \leq_{\text{FUNC}} \lambda \text{obj}:\tau_2 \lambda \tilde{P}_2.f_2.$$

Proof. The lemma follows from Definitions 5.7, 5.4, and 5.6. \square

In the remainder of this chapter, we prove the following theorem.

Theorem 5.9 Equality of functional forms is decidable. \square

Combining this theorem with Definitions 5.4, 5.6, and 5.7, we can conclude that both equality relations and both subfunction relations are decidable.

5.2 Decidability of equality

In this section, we prove Theorem 5.9. More precisely, we prove that there is a decision procedure for equality of functional forms.

First, we prove a number of lemmas. The following lemma states that, for every set of functions with domain string^m and codomain string, there is at least one input on which all the functions in the set differ.

Lemma 5.10 Let H_1, \dots, H_n be functions with domain string^m and codomain string, where the only operation is concatenation. Then:

$$\begin{aligned} \forall i, j \in \{1, \dots, n\} [i \neq j \Rightarrow \tilde{H}_i \neq \tilde{H}_j] &\Rightarrow \\ \exists \vec{x} \in \text{string}^m [\forall i, j \in \{1, \dots, n\} [i \neq j \Rightarrow \tilde{H}_i(\vec{x}) \neq \tilde{H}_j(\vec{x})]], \end{aligned}$$

where \tilde{H} is the normal form of H according to the following rewrite rules [31]:

$$\begin{aligned} \epsilon + w &\longrightarrow w \\ w + \epsilon &\longrightarrow w \\ c + d &\longrightarrow \text{concat}(c, d), \end{aligned}$$

where ϵ is the empty string, w is a string expression, c and d are string constants, and $\text{concat}(c, d) = \mathcal{E}(+, c, d)$.

Proof. Let N be the length of \vec{x} and L be the length of the longest string constant in any of the \tilde{H}_i . Let K be the maximum of N and $\lceil 2 \log L \rceil$, $\pi(i)$ be 2^{K+i} , and $\rho(i)$ be 2^{N-i} . Furthermore, let a and b be different string constants of length 1. For every $i \in \{1, \dots, N\}$, define:

$$v_i = (\text{concat}(a^{\pi(i)}, b^{\pi(i)}))^{\rho(i)},$$

where $c^0 = \epsilon$ and $c^{k+1} = \text{concat}(c, c^k)$. For every pair of string constants c and d of length at most L , $\text{concat}(c, v_i)$ is a prefix of $\text{concat}(d, v_j)$ if and only if $i = j$ and $c = d$. Let \vec{v} be v_1, \dots, v_N . Then:

$$\forall i, j \in \{1, \dots, n\} [\tilde{H}_i \neq \tilde{H}_j \Rightarrow \tilde{H}_i(\vec{v}) \neq \tilde{H}_j(\vec{v})].$$

□

The following lemma states that, for every basic type B and every set of functions with domain B^m and codomain B , there is an input on which all the functions in the set differ.

Lemma 5.11 Let H_1, \dots, H_n be functions with domain B^m and codomain B for some $B \in BTypes$, where the only operations are the operations of Definition 5.1. Then:

$$\begin{aligned} \forall i, j \in \{1, \dots, n\} [i \neq j \Rightarrow H_i \neq H_j] &\Rightarrow \\ \exists \vec{x} \in B^m [\forall i, j \in \{1, \dots, n\} [i \neq j \Rightarrow H_i(\vec{x}) \neq H_j(\vec{x})]]. \end{aligned}$$

Proof.

Case 1: H_1, \dots, H_n are oid-valued functions. The lemma follows from the fact that the body of an oid-valued function is a variable.

Case 2: H_1, \dots, H_n are polynomials with integer variables and coefficients. For every $i, j \in \{1, \dots, n\}$, define $D_{(i,j)}$ to be $H_i - H_j$. Then:

$$\begin{aligned} & \forall i, j \in \{1, \dots, n\} [i \neq j \Rightarrow H_i \neq H_j] \Rightarrow \\ & \forall i, j \in \{1, \dots, n\} [i \neq j \Rightarrow D_{(i,j)} \neq 0] \Rightarrow \\ & \Pi_{i \neq j} D_{(i,j)} \neq 0 \Rightarrow \\ & \exists \vec{x} [(\Pi_{i \neq j} D_{(i,j)})(\vec{x}) \neq 0] \Rightarrow \\ & \exists \vec{x} [\Pi_{i \neq j} (D_{(i,j)}(\vec{x})) \neq 0] \Rightarrow \\ & \exists \vec{x} [\forall i, j \in \{1, \dots, n\} [i \neq j \Rightarrow D_{(i,j)}(\vec{x}) \neq 0]] \Rightarrow \\ & \exists \vec{x} [\forall i, j \in \{1, \dots, n\} [i \neq j \Rightarrow H_i(\vec{x}) \neq H_j(\vec{x})]]. \end{aligned}$$

Case 3: H_1, \dots, H_n are rational functions with rational variables and coefficients, all different from the function that is undefined for every input. That is, for every $i \in \{1, \dots, n\}$, $H_i = F_i/G_i$, where F_i and $G_i \neq 0$ are polynomials with rational variables and coefficients. Let H be $G_1 \times \dots \times G_n$ and, for every $i \in \{1, \dots, n\}$, define:

$$\tilde{H}_i = F_i \times G_1 \times \dots \times G_{i-1} \times G_{i+1} \times \dots \times G_n.$$

Then $H_i = \tilde{H}_i/H$. Furthermore, for every $i, j \in \{1, \dots, n\}$, let $D_{(i,j)}$ be $\tilde{H}_i - \tilde{H}_j$. Then:

$$\begin{aligned} & \forall i, j \in \{1, \dots, n\} [i \neq j \Rightarrow H_i \neq H_j] \wedge H \neq 0 \Rightarrow \\ & \forall i, j \in \{1, \dots, n\} [i \neq j \Rightarrow \tilde{H}_i \neq \tilde{H}_j] \wedge H \neq 0 \Rightarrow \\ & \forall i, j \in \{1, \dots, n\} [i \neq j \Rightarrow D_{(i,j)} \neq 0] \wedge H \neq 0 \Rightarrow \\ & (\Pi_{i \neq j} D_{(i,j)}) \times H \neq 0 \Rightarrow \\ & \exists \vec{x} [(\Pi_{i \neq j} D_{(i,j)}) \times H](\vec{x}) \neq 0] \Rightarrow \\ & \exists \vec{x} [\Pi_{i \neq j} (D_{(i,j)}(\vec{x})) \times H(\vec{x}) \neq 0] \Rightarrow \\ & \exists \vec{x} [\forall i, j \in \{1, \dots, n\} [i \neq j \Rightarrow D_{(i,j)}(\vec{x}) \neq 0] \wedge H(\vec{x}) \neq 0] \Rightarrow \\ & \exists \vec{x} [\forall i, j \in \{1, \dots, n\} [i \neq j \Rightarrow \tilde{H}_i(\vec{x}) \neq \tilde{H}_j(\vec{x})] \wedge H(\vec{x}) \neq 0] \Rightarrow \\ & \exists \vec{x} [\forall i, j \in \{1, \dots, n\} [i \neq j \Rightarrow H_i(\vec{x}) \neq H_j(\vec{x})]]. \end{aligned}$$

Case 4: H_1, \dots, H_n are string-valued functions. Then the lemma follows from lemma 5.10 and the fact that a string-valued function is extensionally equal to its normal form. \square

To extend Lemma 5.11 to general functional forms, we need to define the set of component types of a type and the set of functional forms of a component type.

Definition 5.12 Let σ be a type. The set of component types of σ is defined as $\text{comps}(\sigma, \emptyset)$, where

$$\begin{aligned}
comps(t, V) &= \{(t, V)\} \\
comps(B, V) &= \{(B, V)\} \\
comps(\{v\}, V) &= \{(\{v\}, V)\} \cup comps(v, V) \\
comps(< l_1 : \tau_1, \dots, l_n : \tau_n >, V) &= \\
&\quad \{(< l_1 : \tau_1, \dots, l_n : \tau_n >, V)\} \cup comps(v_1, V) \cup \dots \cup comps(v_n, V) \\
comps(\mu t. \alpha, V) &= \{(\mu t. \alpha, V)\} \cup comps(\alpha, V \cup \{\mu t. \alpha\}).
\end{aligned}$$

□

The set of functional forms of a component type is given by the following recursive definition.

Definition 5.13 Let τ and σ be types, Var' be a finite subset of Var , $Expr'$ be a finite subset of

$$\bigcup \{expr(\tau, B, V) \mid B \in BTypes \wedge (B, V) \in comps(\sigma, \emptyset)\},$$

and $Path'$ be a finite subset of

$$\bigcup \{bexpr(\tau, v, V) \mid (v, V) \in comps(\sigma, \emptyset)\}.$$

For $i \in \mathbb{N}$ and $(v, V) \in comps(\sigma, \emptyset)$, the set of expressions on level i with codomain v and environment V , denoted by $expr_i(\tau, v, V)$, is recursively defined as follows:

$$\begin{aligned}
expr_0(\tau, t, V) &= expr(\tau, t, V) \cap Var' \text{ if } t \in TypeVar \\
expr_0(\tau, B, V) &= expr(B, V) \cap Expr' \text{ if } B \in BTypes \\
expr_0(\tau, v, V) &= expr(\tau, v, V) \cap Path' \text{ if } v \in Types - (TypeVar \cup BTypes) \\
expr_{i+1}(\tau, t, V) &= expr_i(\tau, t, V) \text{ if } t \in TypeVar \\
expr_{i+1}(\tau, B, V) &= expr_i(\tau, B, V) \text{ if } B \in BTypes \\
expr_{i+1}(\tau, \{v\}, V) &= expr_i(\tau, \{v\}, V) \cup \\
&\quad \{e \cup \{e_1, \dots, e_n\} \mid e \in expr_i(\tau, \{v\}, V) \wedge \\
&\quad \forall j \in \{1, \dots, n\} [e_j \in expr_i(\tau, v_j, V)]\} \\
expr_{i+1}(\tau, < l_1 : v_1, \dots, l_n : v_n >, V) &= expr_i(\tau, < l_1 : v_1, \dots, l_n : v_n >, V) \cup \\
&\quad \{< l_1 = e_1, \dots, l_n = e_n > \mid \forall j \in \{1, \dots, n\} [e_j \in expr_i(\tau, v_j, V)]\} \\
expr_{i+1}(\tau, \mu t. \alpha, V) &= expr_i(\tau, \mu t. \alpha, V) \cup \\
&\quad \{\mu x. (e_0[x_1 \setminus e_1, \dots, x_n \setminus e_n]) \mid x \in Var_t \wedge e_0 \in expr_i(\tau, \alpha, V \cup \{\mu t. \alpha\}) \wedge \\
&\quad FV(e_0) \cap Var_t = \{x_1, \dots, x_n\} \wedge \\
&\quad \forall j \in \{1, \dots, n\} [e_j \in Var_t \cup expr_i(\tau, \mu t. \alpha, V) \wedge x \notin BV(e_j)]\}.
\end{aligned}$$

Finally, the set of functional forms on level i with codomain v and environment V , denoted by $\mathcal{F}_i(\tau, v, V)$, is defined by:

$$\mathcal{F}_i(\tau, v, V) = \{\lambda obj : \tau \lambda \vec{p} : \vec{P}. e \mid k \in \mathbb{N} \wedge \vec{p} \in \mathcal{L}^k \wedge \vec{P} \in Btypes^k \wedge e \in expr_i(\tau, v, V)\}.$$

□

To extend Lemma 5.11 to general functional forms, for every set of basic functions, we construct a special instance on which all derived functions disagree.

Construction 5.14 Let τ and σ be types. For every $(v, V) \in \text{comps}(\sigma, \emptyset)$, the set of paths of type v w.r.t. environment V , denoted by $\text{paths}(v, V)$, is inductively defined as follows:

1. if $\text{path} \in \text{Path}' \cap \text{expr}_0(\tau, v, V)$,
then $\text{path} \in \text{paths}(v, V)$
2. if $\text{path} \in \text{expr}_0(\tau, \langle l_1 : v_1, \dots, l_n : v_n \rangle, V)$,
then $\text{path}.l_i \in \text{paths}(v_i, V)$ for every $i \in \{1, \dots, n\}$
3. if $\text{path} \in \text{expr}_0(\tau, \mu t. \langle l_1 : v_1, \dots, l_n : v_n \rangle, V)$,
where $t \notin FV(\langle l_1 : v_1, \dots, l_n : v_n \rangle)$,
then $\text{path}.l_i \in \text{paths}(v_i, V)$ for every $i \in \{1, \dots, n\}$.

Let the set of basic functions on level 0, denoted by $B\text{Functions}_0$, be given by:

$$B\text{Functions}_0 = \{ \lambda \text{obj} : \tau \lambda \vec{p} : \vec{P}. f \mid B \in B\text{Types} \wedge (B, V) \in \text{comps}(\sigma, \emptyset) \wedge \\ k \in \mathbb{N} \wedge \vec{p} \in \mathcal{L}^k \wedge \vec{P} \in B\text{types}^k \wedge \\ f \in \text{paths}(B, V) \cup \text{expr}_0(\tau, B, V) \}.$$

According to Lemma 5.11, there is a vector $\vec{x} = (x_0, x_1, \dots, x_k)$ in $\llbracket \tau \rrbracket \times \llbracket \tau_1 \rrbracket \times \dots \times \llbracket \tau_k \rrbracket$, such that:

$$\forall F, G \in B\text{Functions}_0 [F \neq_{\text{FUNC}} G \Rightarrow F_{\square}(\vec{x}) \neq_D G_{\square}(\vec{x})].$$

Let $\vec{d} = (d_0, d_1, \dots, d_k)$ be this vector and change the vector as follows. Let the set of set-valued functions on level 0, denoted by $S\text{Functions}_0$, be given by:

$$S\text{Functions}_0 = \{ \lambda \text{obj} : \tau \lambda \vec{p} : \vec{P}. f \mid k \in \mathbb{N} \wedge \vec{p} \in \mathcal{L}^k \wedge \vec{P} \in B\text{types}^k \wedge f \in S\text{Paths} \},$$

where

$$S\text{Paths} = \bigcup \{ \text{paths}(\{v\}, V) \mid (\{v\}, V) \in \text{comps}(\sigma, \emptyset) \}$$

and the set of recursive record-valued functions on level 0, denoted by $R\text{Functions}_0$, be given by:

$$R\text{Functions}_0 = \{ \lambda \text{obj} : \tau \lambda \vec{p} : \vec{P}. f \mid k \in \mathbb{N} \wedge \vec{p} \in \mathcal{L}^k \wedge \vec{P} \in B\text{types}^k \wedge f \in R\text{Paths} \},$$

where

$$R\text{Paths} = \bigcup \{ \text{paths}(\mu t. \alpha, V) \mid (\mu t. \alpha, V) \in \text{comps}(\sigma, \emptyset) \wedge t \in FV(\alpha) \}.$$

For every path $\text{obj.path} \in S\text{Paths}_0 \cup R\text{Paths}_0$, choose a unique constant $c_{\text{path}} \in \text{Cons}$, different from any constant in $\{F_{\square}(\vec{d}) \mid F \in B\text{Functions}_0\}$, and extend d_0 in such a way that c_{path} occurs in $d_0.\text{path}$. Let c_0 be d_0 after extension and \vec{c} be (c_0, d_1, \dots, d_k) . Then:

$$\forall F, G \in Functions_0 [F \neq_{FUNC} G \Rightarrow F_{\square}(\vec{c}) \neq_D G_{\square}(\vec{c})],$$

where $Functions_0 = BFunctions_0 \cup SFunctions_0 \cup RFunctions_0$. \square

Now we can extend Lemma 5.11 to general functional forms.

Lemma 5.15 Let τ and σ be types, and \vec{c} be the vector constructed in Construction 5.14. Then:

$$\begin{aligned} \forall i \in \mathbb{N} \forall (v, V) \in comps(\sigma, \emptyset) \\ \forall F, G \in \mathcal{F}_i(v, V) [F \neq_{FUNC} G \Rightarrow F_{\square}(\vec{c}) \neq_D G_{\square}(\vec{c})]. \end{aligned}$$

Proof. The proof is an induction argument on i . Base step: $i = 0$. The lemma follows directly from Construction 5.14.

Induction step: $i = i' + 1$. Let $Paths_0$ be $\bigcup \{paths(v, V) \mid (v, V) \in comps(\sigma, \emptyset)\}$.

Case 1: $F, G \in \mathcal{F}_{i'+1}(\tau, t, V)$, where $t \in TypeVar$. Using the fact that $\mathcal{F}_{i'+1}(\tau, t, V) = \mathcal{F}_{i'}(\tau, t, V)$ and the induction hypothesis, we can deduce that $F(\vec{c}) \neq_D G(\vec{c})$.

Case 2: $F, G \in \mathcal{F}_{i'+1}(\tau, B, V)$, where $B \in BTypes$. The same as case 1.

Case 3: $F, G \in \mathcal{F}_{i'+1}(\tau, \{v\}, V)$. Then:

1. $F = \lambda \vec{q} : \vec{Q}. (f \cup \{f_1, \dots, f_n\})$
2. $G = \lambda \vec{q} : \vec{Q}. (g \cup \{g_1, \dots, g_{n'}\})$

where $n, n' \geq 0$, $f, g \in Paths_0$ and $f_j, g_j \in expr_{i'}(\tau, v, V)$. Now, suppose $F \neq_{FUNC} G$. If $\lambda \vec{q} : \vec{Q}. f \neq_{FUNC} \lambda \vec{q} : \vec{Q}. g$, then it follows from Construction 5.14 that $F(\vec{c}) \neq_D G(\vec{c})$.

If $\lambda \vec{q} : \vec{Q}. f =_{FUNC} \lambda \vec{q} : \vec{Q}. g$, then either:

- $\exists j \in \{1, \dots, n\} \forall j' \in \{1, \dots, n'\} [\lambda \vec{q} : \vec{Q}. f_j \neq_{FUNC} \lambda \vec{q} : \vec{Q}. g_{j'}]$ or
- $\exists j' \in \{1, \dots, n'\} \forall j \in \{1, \dots, n\} [\lambda \vec{q} : \vec{Q}. f_j \neq_{FUNC} \lambda \vec{q} : \vec{Q}. g_{j'}]$.

Using the induction hypothesis, we can deduce that $F(\vec{c}) \neq_D G(\vec{c})$.

Case 4: $F, G \in \mathcal{F}_{i'+1}(\tau, < l_1 : v_1, \dots, l_n : v_n >, V)$. Then $F = \lambda \vec{q} : \vec{Q}. f$ and $G = \lambda \vec{q} : \vec{Q}. g$, where:

1. $f \in Paths_0$ or $f = < l_1 = f_1, \dots, l_n = f_n >$
2. $g \in Paths_0$ or $g = < l_1 = g_1, \dots, l_n = g_n >$

where $f_j, g_j \in expr_{i'}(\tau, v_j, V)$. Now, suppose $F \neq_{FUNC} G$. If $f \in Paths_0$ and $g \in Paths_0$, then it follows from the induction hypothesis that $F(\vec{c}) \neq_D G(\vec{c})$.

If $f = < l_1 = f_1, \dots, l_n = f_n >$ and $g = < l_1 = g_1, \dots, l_n = g_n >$, then:

$$\exists j \in \{1, \dots, n\} [\lambda \vec{q} : \vec{Q}. \vec{Q}. g_j \neq_{FUNC} \lambda \vec{q} : \vec{Q}. \vec{Q}. f_j],$$

because otherwise:

$$\begin{aligned} G &=_{FUNC} \lambda \vec{q} : \vec{Q}. < l_1 = g_1, \dots, l_n = g_n > \\ &=_{FUNC} \lambda \vec{q} : \vec{Q}. < l_1 = f_1, \dots, l_n = f_n > =_{FUNC} F. \end{aligned}$$

From the induction hypothesis, we can deduce that for some j in $\{1, \dots, n\}$:

$$g_j[\vec{q} \setminus \vec{c}] \neq_D f_j[\vec{q} \setminus \vec{c}].$$

Hence, $F(\vec{c}) \neq_D G(\vec{c})$.

If $f \in \text{Paths}_0$ and $g = \langle l_1 = g_1, \dots, l_n = g_n \rangle$, then:

$$\exists j \in \{1, \dots, n\} [\lambda \vec{q} : \vec{Q}.g_j \neq_{\text{FUNC}} \lambda \vec{q} : \vec{Q}.(f.l_j)],$$

because otherwise:

$$\begin{aligned} G &=_{\text{FUNC}} \lambda \vec{q} : \vec{Q}. \langle l_1 = g_1, \dots, l_n = g_n \rangle \\ &=_{\text{FUNC}} \lambda \vec{q} : \vec{Q}. \langle l_1 = f.l_1, \dots, l_n = f.l_n \rangle =_{\text{FUNC}} F. \end{aligned}$$

From Construction 5.14 we can deduce that for some j in $\{1, \dots, n\}$:

$$g_j[\vec{q} \setminus \vec{c}] \neq_D f.l_j[\vec{q} \setminus \vec{c}].$$

Hence, $F(\vec{c}) \neq_D G(\vec{c})$.

For $f = \langle l_1 = f_1, \dots, l_n = f_n \rangle$ and $g \in \text{Paths}_0$, the proof is obtained in the same way.

Case 5: $F, G \in \mathcal{F}_{i'+1}(\tau, \mu t. \langle l_1 : v_1, \dots, l_n : v_n \rangle, V)$, where $t \notin \text{FV}(\langle l_1 : v_1, \dots, l_n : v_n \rangle)$. The same as case 4.

Case 6: $F, G \in \mathcal{F}_{i'+1}(\tau, \mu t. \alpha, V)$, where $t \in \alpha$. Then $F = \lambda \vec{q} : \vec{Q}.f$ and $G = \lambda \vec{q} : \vec{Q}.g$, where

1. $f \in \text{Paths}_0$ or $f = \mu x.f_0[x_1 \setminus f_1, \dots, x_n \setminus f_n]$
2. $g \in \text{Paths}_0$ or $g = \mu y.g_0[y_1 \setminus g_1, \dots, y_{n'} \setminus g_{n'}]$

where $f_0, g_0 \in \text{expr}_{i'}(\tau, \alpha, V)$, $\text{FV}(f_0) \cap \text{Var}_t = \{x_1, \dots, x_n\}$, $\text{FV}(g_0) \cap \text{Var}_t = \{y_1, \dots, y_{n'}\}$, $f_j \in \text{Var}_t \cup \text{expr}_{i'}(\tau, \mu t. \alpha, V)$ and $x \notin \text{BV}(f_j)$ for $j \in \{1, \dots, n\}$, and $g_j \in \text{Var}_t \cup \text{expr}_{i'}(\tau, \mu t. \alpha, V)$ and $y \notin \text{BV}(g_j)$ for $j \in \{1, \dots, n'\}$. Now, suppose $F \neq_{\text{FUNC}} G$. If $f \in \text{Paths}_0$ and $g \in \text{Paths}_0$, then it follows from the induction hypothesis that $F(\vec{c}) \neq_D G(\vec{c})$.

If $f = \mu x.f_0[x_1 \setminus f_1, \dots, x_n \setminus f_n]$ and $g = \mu y.g_0[y_1 \setminus g_1, \dots, y_{n'} \setminus g_{n'}]$, then either there is a bijection h from $\{1, \dots, n\}$ to $\{1, \dots, n'\}$, such that $f_0[x \setminus z, x_1 \setminus y_{h(1)}, \dots, x_n \setminus y_{h(n)}] =_{\text{FUNC}} g_0[y \setminus z]$ or there is not such a bijection, where z is a fresh instance variable. If there is such a bijection h , then:

$$\exists j \in \{1, \dots, n\} [\lambda \vec{q} : \vec{Q}.f_j[x \setminus z] \neq_{\text{FUNC}} \lambda \vec{q} : \vec{Q}.g_{h(j)}[y \setminus z]],$$

because otherwise:

$$\begin{aligned} F &=_{\text{FUNC}} \lambda \vec{q} : \vec{Q}.(\mu x.f_0[x_1 \setminus f_1, \dots, x_n \setminus f_n]) \\ &=_{\text{FUNC}} \lambda \vec{q} : \vec{Q}.(\mu y.g_0[y_1 \setminus g_1, \dots, y_{n'} \setminus g_{n'}]) =_{\text{FUNC}} G. \end{aligned}$$

Let F_j be $\lambda \vec{q} : \vec{Q}.f_j$ and G_j be $\lambda \vec{q} : \vec{Q}.g_j$ for $j \in \{1, \dots, n\}$. Using the induction hypothesis, we can deduce that $F_j(\vec{c}) \neq_D G_{h(j)}(\vec{c})$ for some j in $\{1, \dots, n\}$. Hence:

$$F(\vec{c}) = \mu x. F_0(\vec{c})[x_1 \setminus F_1(\vec{c}), \dots, x_n \setminus F_n(\vec{c})] \\ \neq_D \mu y. G_0(\vec{c})[y_1 \setminus G_1(\vec{c}), \dots, y_n \setminus G_n(\vec{c})] = G(\vec{c}).$$

If there is not such a bijection h , then it follows from the induction hypothesis that $F(\vec{c}) \neq_D G(\vec{c})$.

For both other combinations of f and g , we can deduce $F(\vec{c}) \neq_D G(\vec{c})$ using Construction 5.14. \square

5.2.1 Decision procedure

We prove that, for every pair of types τ and σ , there is a decision procedure for equality of functions in:

$$\{\mathcal{F}_i(\tau, v, V) \mid i \in \mathbb{N} \wedge (v, V) \in \text{comps}(\sigma, \emptyset)\}.$$

The proof is an induction argument on i . Base step: $i = 0$. The proof is an induction argument.

Case 1: $F, G \in \mathcal{F}_0(\tau, t, V)$, where $t \in \text{TypeVar}$. From the fact that the bodies of F and G are instance variables we can deduce that equality of F and G can be tested effectively.

Case 2: $F, G \in \mathcal{F}_0(\tau, \text{oid}, V)$. From the fact that the bodies of F and G are paths we can deduce that equality of F and G can be tested effectively.

Case 3: $F, G \in \mathcal{F}_0(\tau, \text{integer}, V)$. Then F and G are polynomials with integer variables and coefficients. It follows from [35] that equality of F and G can be tested effectively by comparing the coefficients of their normal forms.

Case 4: $F, G \in \mathcal{F}_0(\tau, \text{rational}, V)$. Then F and G are rational functions with rational variables and coefficients. It follows from [35] that equality of F and G can be tested effectively by comparing the coefficients of their normal forms.

Case 5: $F, G \in \mathcal{F}_0(\tau, \text{string}, V)$. Reversing Lemma 5.10 and using $n = 2$ gives us:

$$\forall \vec{x} [\tilde{H}_1(\vec{x}) = \tilde{H}_2(\vec{x})] \Rightarrow \tilde{H}_1 \equiv \tilde{H}_2.$$

Hence:

$$F =_{\text{FUNC}} G \Leftrightarrow \tilde{F} =_{\text{FUNC}} \tilde{G} \Leftrightarrow \tilde{F} \equiv \tilde{G}.$$

It follows that equality of F and G can be tested effectively by comparing their normal forms.

Case 6: $F, G \in \mathcal{F}_0(\tau, v, V)$, where $v \in \text{Types} - (\text{TypeVar} \cup \text{BTypes})$. From the fact that the bodies of F and G are paths we can deduce that equality of F and G can be tested effectively.

Induction step: $i = i' + 1$. Case 1: $F, G \in \mathcal{F}_{i'+1}(\tau, t, V)$, where $t \in \text{TypeVar}$. From the fact that $\mathcal{F}_{i'+1}(\tau, t, V) = \mathcal{F}_{i'}(t, V)$ and the induction hypothesis we can deduce that equality of F and G can be tested effectively.

Case 2: $F, G \in \mathcal{F}_{i'+1}(\tau, B, V)$, where $B \in \text{BTypes}$. The same as case 1.

Case 3: $F, G \in \mathcal{F}_{i'+1}(\tau, \{v\}, V)$. Then:

1. $F = \lambda \vec{q} : \vec{Q}.(f \cup \{f_1, \dots, f_n\})$
2. $G = \lambda \vec{q} : \vec{Q}.(g \cup \{g_1, \dots, g_{n'}\})$

where $n, n' \geq 0$, and $f, g \in Paths_0$ and $f_j, g_j \in expr_{i'}(\tau, v, V)$. It follows from Lemma 5.15 that $F =_{FUNC} G$ if and only if:

1. $f = g$
2. $\forall j \in \{1, \dots, n\} \exists j' \in \{1, \dots, n'\} [\lambda \vec{q} : \vec{Q}.f_j =_{FUNC} \lambda \vec{q} : \vec{Q}.g_{j'}]$ or
3. $\forall j' \in \{1, \dots, n'\} \exists j \in \{1, \dots, n\} [\lambda \vec{q} : \vec{Q}.f_j =_{FUNC} \lambda \vec{q} : \vec{Q}.g_{j'}]$.

Using the induction hypothesis, we can deduce that equality of F and G can be tested effectively.

Case 4: $F, G \in \mathcal{F}_{i'+1}(\tau, < l_1 : v_1, \dots, l_n : v_n >, V)$. Then $F = \lambda \vec{q} : \vec{Q}.f$ and $G = \lambda \vec{q} : \vec{Q}.g$, where:

1. $f \in Paths_0$ or $f = < l_1 = f_1, \dots, l_n = f_n >$
2. $g \in Paths_0$ or $g = < l_1 = g_1, \dots, l_n = g_n >$

where $f_j, g_j \in expr_{i'}(\tau, v_j, V)$. If $f \in Paths_0$ and $g \in Paths_0$, then $F =_{FUNC} G$ if and only if $f = g$, which can be tested effectively.

If $f = < l_1 = f_1, \dots, l_n = f_n >$ and $g = < l_1 = g_1, \dots, l_n = g_n >$, then $F =_{FUNC} G$ if and only if:

$$\forall j \in \{1, \dots, n\} [\lambda \vec{q} : \vec{Q}.g_j =_{FUNC} \lambda \vec{q} : \vec{Q}.f_j].$$

Using the induction hypothesis, we can deduce that equality of F and G can be tested effectively.

If $f \in Paths_0$ and $g = < l_1 = g_1, \dots, l_n = g_n >$, then $F =_{FUNC} G$ if and only if:

$$\forall j \in \{1, \dots, n\} [\lambda \vec{q} : \vec{Q}.g_j =_{FUNC} \lambda \vec{q} : \vec{Q}.(f.l_j)].$$

Using the induction hypothesis, we can deduce that equality of F and G can be tested effectively.

For $f = < l_1 = f_1, \dots, l_n = f_n >$ and $g \in Paths_0$, the proof is obtained in the same way.

Case 5: $F, G \in \mathcal{F}_{i'+1}(\tau, \mu t. < l_1 : v_1, \dots, l_n : v_n >, V)$, where $t \notin FV(< l_1 : v_1, \dots, l_n : v_n >)$. The same as case 4.

Case 6: $F, G \in \mathcal{F}_{i'+1}(\tau, \mu t.\alpha, V)$, where $t \in \alpha$. Then $F = \lambda \vec{q} : \vec{Q}.f$ and $G = \lambda \vec{q} : \vec{Q}.g$, where

1. $f \in Paths_0$ or $f = \mu x.f_0[x_1 \setminus f_1, \dots, x_n \setminus f_n]$
2. $g \in Paths_0$ or $g = \mu y.g_0[y_1 \setminus g_1, \dots, y_{n'} \setminus g_{n'}]$,

where $f_0, g_0 \in expr_{i'}(\tau, \alpha, V)$, $FV(f_0) \cap Var_t = \{x_1, \dots, x_n\}$, $FV(g_0) \cap Var_t = \{y_1, \dots, y_{n'}\}$, $f_j \in Var_t \cup expr_{i'}(\tau, \mu t.\alpha, V)$ and $x \notin BV(f_j)$ for $j \in \{1, \dots, n\}$, and $g_j \in Var_t \cup expr_{i'}(\tau, \mu t.\alpha, V)$ and $y \notin BV(g_j)$ for $j \in \{1, \dots, n'\}$.

If $f \in Paths_0$, then $F =_{FUNC} G$ if and only if $f = g$, which can be tested effectively.

If $f = \mu x.f_0[x_1 \setminus f_1, \dots, x_n \setminus f_n]$, then $F =_{FUNC} G$ if and only if $g = \mu y.g_0[y_1 \setminus g_1, \dots, y_{n'} \setminus g_{n'}]$ and there is a bijection h from $\{1, \dots, n\}$ to $\{1, \dots, n'\}$, such that

$$\forall j \in \{1, \dots, n\} [\lambda \vec{q} : \vec{Q}.f_j[x \setminus z] =_{FUNC} \lambda \vec{q} : \vec{Q}.g_{h(j)}[y \setminus z]],$$

where z is a fresh type variable. Using the induction hypothesis, we can deduce that equality of F and G can be tested effectively.

The core of the decision procedure consists of normalisation of basic functions and syntactic comparison of the normalised basic functions, which can both be done in polynomial time.

In fact, we can normalise general functional forms by replacing every basic expression by its normal form and replacing every expression of the form $\langle l_1 = path.l_1, \dots, l_n = path.l_n \rangle$ by $path$ if $\tau.path = \langle l_1:v_1, \dots, l_n:v_n \rangle$ for some (v_1, \dots, v_n) and replacing every expression of the form $\mu x. \langle l_1 = path.l_1, \dots, l_n = path.l_n \rangle$ by $path$ if $\tau.path = \mu t. \langle l_1:v_1, \dots, l_n:v_n \rangle$ for some (v_1, \dots, v_n) , such that $t \notin FV(\langle l_1:v_1, \dots, l_n:v_n \rangle)$. After normalisation, we can test for equality by syntactically comparing the normalised functional forms. Both normalisation and syntactic comparison can be done in polynomial time. Hence, we can conclude that testing for equality of functional forms is polynomial time decidable.

Chapter 6

Schema comparison

In this chapter, we want to compare schemas by their real world semantics. The real world semantics of a schema is given by the sequence of states of the part of the real world described by the schema. The attributes, formalised by types, describe the states of real world objects and the update methods, formalised by functions, describe the transitions between states. It follows that we can compare classes by comparing types and functions ‘separately’. To compare classes, we combine the subtype relation and the subfunction relation defined in the previous chapters into a (synthetic) subclass relation.

A class is a subclass of another class if the underlying type of the first class is a subtype of the underlying type of the second class and the functional forms of the first class are subfunctions of the functional forms of the second class.

Definition 6.1 Let H be a well-defined class hierarchy and let C_1 and C_2 be classes in H . Then C_1 is a subclass of C_2 , denoted by $C_1 \preceq_{CLASS} C_2$, if and only if:

1. $type(C_1) \preceq_{TYPE} type(C_2)$
2. $\forall f_2 \in func(C_2) \exists f_1 \in func(C_1) [f_1 \leq_{FUNC}^w f_2]$

where $func(C) = \{func(C, meth) \mid meth \in meths(C)\}$. \square

An implicit assumption we make is that $BC \subseteq Cons$. The subclass relation compares classes by the following characteristics: the structure of the objects in the class extensions (subtype relation) and the way these objects are manipulated (subfunction relation). These characteristics can be regarded as abstract semantics, where classes are semantically equal if the objects in the class extensions have the same structure and are manipulated in the same way. Since abstract semantics are used to compare classes, rather than real world semantics, the subclass order is called synthetic.

In the remainder of this chapter, we extend the subclass relation in order to cope with classes that have the same abstract semantics, except that their attribute names are different. For that purpose, we define subclass morphisms based on subtype morphisms.

6.1 Comparison of attributes

In this section, we introduce subtype morphisms. To define subtype morphisms, we have to define type morphisms, faithfulness with respect to classes, and faithfulness with respect to attributes. definitions. A type morphism is a tree homomorphism between the trees representing the types.

Definition 6.2 Let τ_1 and τ_2 be types. A type morphism from τ_1 to τ_2 is a tree homomorphism from $estruc(\tau_1)$ to $estruc(\tau_2)$. \square

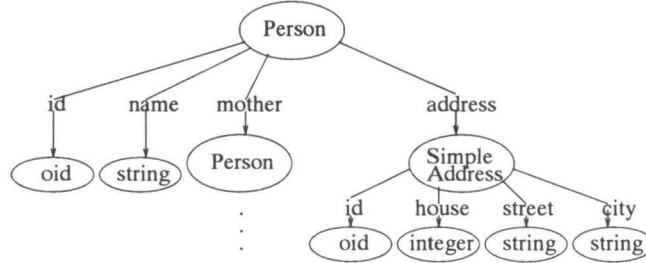
A type morphism between underlying types of classes is faithful with respect to classes if it maps class names to class names.

Definition 6.3 Let H be a well-defined class hierarchy, C_1 and C_2 be classes in H , and φ be a type morphism from $type(C_1)$ to $type(C_2)$. Then φ is faithful w.r.t. classes if and only if for every node n in $estruc(type(C_1))$ the following holds:

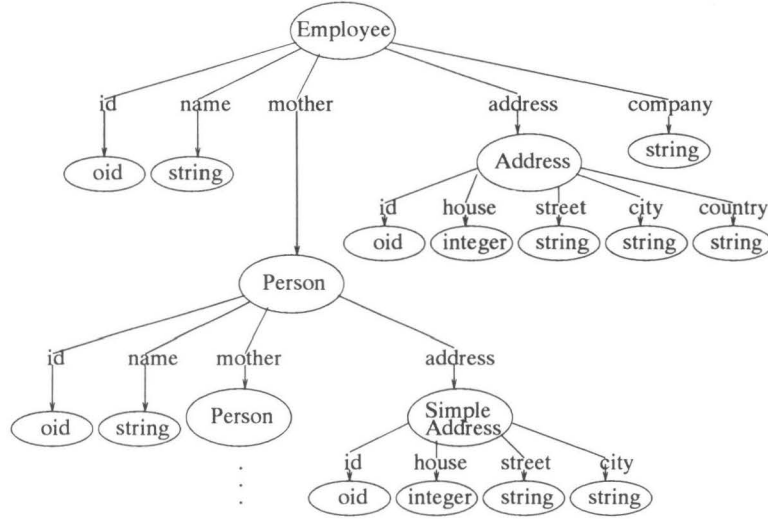
$$label(n) \in TypeVar \Rightarrow label(\varphi(n)) \in TypeVar.$$

\square

Example 6.4 Let C_P be class 'Person' and C_E be class 'Employee' in the class hierarchy of Example 3.9. Then $estruc(type(C_P))$ is given by:



and $estruc(type(C_E))$ is given by:



Any type morphism from $type(C_P)$ to $type(C_E)$ that maps nodes labeled t_P to nodes labeled t_E or t_P and nodes labeled t_S to nodes labeled t_A or t_S is faithful w.r.t. classes. \square

A type morphism between underlying types of classes is faithful with respect to attributes if it maps attribute names in one class to attributes names in another class consistently.

Definition 6.5 Let H be a well-defined class hierarchy, C_1 and C_2 be classes in H , and φ be a type morphism from $type(C_1)$ to $type(C_2)$. Then φ is faithful w.r.t. attributes if and only if for every node n_1 in $estruc(type(C_1))$ with outgoing arrow r_1 and every node n_2 in $estruc(type(C_1))$ with outgoing arrow r_2 the following holds:

$$\begin{aligned}
 & (label(n_1) = label(n_2) \in TypeVar \\
 & \quad \wedge label(\varphi(n_1)) = label(\varphi(n_2)) \\
 & \quad \wedge label(r_1) = label(r_2)) \Rightarrow label(\varphi(r_1)) = label(\varphi(r_2)).
 \end{aligned}$$

\square

Example 6.6 Let C_P be class ‘Person’ and C_E be class ‘Employee’ in the class hierarchy of Example 3.9. The type morphism from $type(C_P)$ to $type(C_E)$ that preserves labels, except class names, is faithful w.r.t. attributes. If all arrows labeled ‘street’ are mapped to arrows labeled ‘city’, then the graph homomorphism is still faithful w.r.t. attributes. However, if only one of the arrows labeled ‘street’ is mapped to an arrow labeled ‘city’, then the graph homomorphism is not faithful any more. \square

Now we can define subtype morphisms between underlying types of classes.

Definition 6.7 Let H be a well-defined class hierarchy, C_1 and C_2 be classes in H , and φ be a type morphism from $\text{type}(C_1)$ to $\text{type}(C_2)$. Then φ is a subtype morphism if and only if it:

1. is injective
2. preserves labels, except class names and attribute names
3. is faithful with respect to classes.

□

Example 6.8 Let C_{P1} be class ‘Person1’ and C_{P2} be class ‘Person2’ of Example 3.10. Let φ be the graph homomorphism from $\text{struc}(C_{P1})$ to $\text{struc}(C_{P2})$ that maps the node labeled ‘Person1’ to the node labeled ‘Person2’, maps the arrow labeled ‘addr’ to the arrow labeled ‘address’, and preserves the labels of the other nodes and arrows. Then φ is injective, maps the root to the root, maps labels to themselves, except class names and attribute names, and is faithful w.r.t. classes and attributes. □

Lemma 6.9 Let H be a well-defined class hierarchy. Let C_1 and C_2 be classes in H . If $\text{type}(C_1) \preceq_{\text{TYPE}} \text{type}(C_2)$, then there is a subtype morphism from $\text{type}(C_1)$ to $\text{type}(C_2)$.

Proof. Suppose $\text{type}(C_1) \preceq_{\text{ext}} \text{type}(C_2)$. According to Theorem 4.51 there is an injective tree homomorphism from $\text{struc}(\text{type}(C_2))$ to $\text{struc}(\text{type}(C_1))$ that preserves labels. Using Lemma 4.36, we can conclude that there is an injective tree homomorphism from $\text{estruc}(\text{type}(C_2))$ to $\text{estruc}(\text{type}(C_1))$ that preserves labels, except class names, and is faithful w.r.t. classes. □

Lemma 6.10 If there exists a subtype morphism from $\text{type}(C_1)$ to $\text{type}(C_2)$, then there exists a subtype morphism from $\text{type}(C_1)$ to $\text{type}(C_2)$ that is faithful w.r.t. attributes.

Proof. We proof this by structural induction.

Basic step: Let φ be a subtype morphism from $\text{type}(B)$ to $\text{type}(T)$. Then it follows immediately that φ is faithful w.r.t. attributes.

Induction step for sets: Let φ be a subtype morphism from $\text{type}(\{U\})$ to $\text{type}(\{U'\})$. By induction, there is a subtype morphism ψ from $\text{type}(U)$ to $\text{type}(U')$ that is faithful w.r.t. attributes. Then ψ can be extended to a subtype from $\text{type}(\{U\})$ to $\text{type}(\{U'\})$ that is faithful w.r.t. attributes.

Induction step for records: Let φ be a subtype morphism from $\text{type}(\langle l_i : U_i \mid i \in I \rangle)$ to $\text{type}(\langle l_i : U'_i \mid i \in I' \rangle)$. By induction, there are subtype morphisms φ_i from $\text{type}(U_i)$ to $\text{type}(U'_i)$ that are faithful w.r.t. attributes. If $\text{estruc}(\text{type}(C))$ is a subtree of both $\text{estruc}(\text{type}(U_i))$ and $\text{estruc}(\text{type}(U_j))$, for some class C , and

both φ_i and φ_j map $estruc(type(C))$ to the same $estruc(type(C'))$ differently, then choose one possibility to map $estruc(type(C))$ to $estruc(type(C'))$ and adapt the subtype morphisms accordingly. The resulting subtype morphisms can be combined into a subtype morphism from $type(< l_i : U_i \mid i \in I >)$ to $type(< l_i : U'_i \mid i \in I' >)$ that is faithful w.r.t. attributes.

Induction step for classes: Let φ be a subtype morphism from $type(C)$ to $type(C')$, $atts(C)$ be $\{a_1 : T_1, \dots, a_k : T_k\}$, and $atts(C')$ be $\{a'_1 : T'_1, \dots, a'_{k'} : T'_{k'}\}$. Then there are subtype morphisms φ_i from $type(T_i)$ to $type(T'_{f(i)})$, where $f(i) = j$ if φ maps attribute a_i to attribute a'_j . Let $estruc_i$ be obtained from $estruc(type(T_i))$ by removing subtrees $estruc(type(C))$ whose image under φ_i is $estruc(type(C'))$ and $estruc'_i$ be obtained from $estruc(type(T'_{f(i)}))$ by removing subtrees $estruc(type(C'))$ which are the images of $estruc(type(C))$ under φ_i . By induction, the projection of φ_i onto $estruc_i$ and $estruc'_i$, denoted by $\tilde{\varphi}_i : estruc_i \rightarrow estruc'_i$, is a subtype morphism that is faithful w.r.t. attributes. Let $estruc$ be obtained from $estruc(type(C))$ by replacing every $estructype((T_i))$ by $estruc_i$ and $estruc'$ be obtained from $estruc(type(C'))$ by replacing every $estruc(type(T'_{f(i)}))$ by $estruc'_i$. As for set types, the subtype morphisms $\tilde{\varphi}_i$ can be combined into a subtype morphism ψ from $estruc$ to $estruc'$ that is faithful w.r.t. attributes. Finally, ψ can be extended to a subtype morphism from $estruc(type(C))$ to $estruc(type(C'))$ that is faithful w.r.t. attributes by extending $estruc$ to $estruc(type(C))$, extending $estruc'$ to $estruc(type(C'))$, and extending ψ accordingly. \square

6.2 Comparison of methods

To define subclass morphisms, we have to define specialisations of functional forms and generalisations of methods. The specialisation of a functional form of a method is obtained by extending it to the type of a subclass.

Definition 6.11 Let H be a well-defined class hierarchy, C_1 and C_2 be classes in H , and φ be a subtype morphism from $type(C_1)$ to $type(C_2)$. Furthermore, let $meth$ be a method in $meths(C_1)$, and $F = \lambda obj:\tau \lambda P.f$ be $func(C_1, meth)$. Functional form $S_\varphi(F)$ is defined by:

$$\lambda obj:\varphi(\tau) \lambda P. S_\varphi(f, empty),$$

where $empty$ denotes the empty path and:

$$\begin{aligned} S_\varphi(x, p) &= x \\ &\text{if } x \in Var \\ S_\varphi(l, p) &= l \\ &\text{if } l \in \mathcal{L} \\ S_\varphi(obj.label(r_1) \dots label(r_n), p) &= obj.label(\varphi(r_1)) \dots label(\varphi(r_n)) \end{aligned}$$

$$\begin{aligned}
& \text{if } r_1 \cdots r_n \text{ is a path in } \text{estruc}(\text{type}(C_1)) \\
& \mathcal{S}_\varphi(b, p) = b \text{ if } b \in \text{Cons} \\
& \mathcal{S}_\varphi(g_1 \theta g_2, p) = \mathcal{S}_\varphi(g_1, p) \theta \mathcal{S}_\varphi(g_2, p) \\
& \quad \text{if } \theta \in \{+, -, \times, \div\} \\
& \mathcal{S}_\varphi(g \cup \{g_1, \dots, g_n\}, p) = \\
& \quad \mathcal{S}_\varphi(g, p) \cup \{\mathcal{S}_\varphi(g_1, p. \in), \dots, \mathcal{S}_\varphi(g_n, p. \in)\} \\
& \mathcal{S}_\varphi(< l_1 = g_1, \dots, l_n = g_n >, p) = \\
& \quad < \text{last}(\varphi(p.l_1)) = \mathcal{S}_\varphi(g_1, p.l_1), \dots, \text{last}(\varphi(p.l_n)) = \mathcal{S}_\varphi(g_n, p.l_n) > \\
& \mathcal{S}_\varphi(\mu x.e, p) = \mu x. \mathcal{S}_\varphi(e, p)
\end{aligned}$$

and $\text{last}(l'_1 \cdots l'_n) = l'_n$. \square

The generalisation of a method is obtained by projecting it on a superclass.

Definition 6.12 Let H be a well-defined class hierarchy, C_1 and C_2 be classes in H , and φ be a subtype morphism from $\text{type}(C_1)$ to $\text{type}(C_2)$. Since φ is injective, φ^{-1} is a partial tree homomorphism from $\text{estruc}(\text{type}(C_2))$ to $\text{estruc}(\text{type}(C_1))$. Furthermore, let $m(P) = E$ be a method in $\text{meths}(C_2)$. The generalisation of body E , denoted by $\varphi^{-1}(E)$, is defined as follows:

$$\begin{aligned}
& \varphi^{-1}(E_1; E_2) = \varphi^{-1}(E_1); \varphi^{-1}(E_2) \\
& \varphi^{-1}(d := s) = \varphi^{-1}(d) := \varphi^{-1}(s) \\
& \varphi^{-1}(\text{insert}(s, d)) = \text{insert}(\varphi^{-1}(s), \varphi^{-1}(d)) \\
& \varphi^{-1}(b) = b \text{ if } b \in BC \\
& \varphi^{-1}(p) = p \\
& \quad \text{if } p \text{ is the name of a parameter in } P \\
& \varphi^{-1}(\text{self}) = \text{self} \\
& \varphi^{-1}(\text{label}(r_1) \cdots \text{label}(r_n)) = \text{label}(\varphi^{-1}(r_1)) \cdots \text{label}(\varphi^{-1}(r_n)) \\
& \quad \text{if } r_1 \cdots r_n \text{ is a path in } \text{estruc}(\text{type}(C_2)) \\
& \quad \text{and, for every } i \in \{1, \dots, n\}, \varphi^{-1}(r_i) \text{ exists} \\
& \varphi^{-1}(\text{label}(r_1) \cdots \text{label}(r_n)) = \perp \\
& \quad \text{if } r_1 \cdots r_n \text{ is a path in } \text{estruc}(\text{type}(C_2)) \\
& \quad \text{and, for some } i \in \{1, \dots, n\}, \varphi^{-1}(r_i) \text{ does not exist} \\
& \varphi^{-1}(e_1 \theta e_2) = \varphi^{-1}(e_1) \theta \varphi^{-1}(e_2).
\end{aligned}$$

Finally, the generalisation of $m(P) = E$, denoted by $\varphi^{-1}(m(P) = E)$, is defined as:

$$m(\tilde{P}) = \tilde{E},$$

where \tilde{E} is obtained from $\varphi^{-1}(E)$ by

1. removing any assignment $d := s$, such that d contains \perp or s contains \perp or $d := s$ is not well-defined in C_1

2. removing any assignment $\mathbf{insert}(s, d)$, such that s contains \perp or d contains \perp or $\mathbf{insert}(s, d)$ is not well-defined in C_1

and \tilde{P} is obtained from P by removing parameters that do not occur in \tilde{E} . \square

Example 6.13 Let C_{P_1} be class ‘Person1’ and C_{P_2} be class ‘Person2’ of 3.10. Let φ be the subtype morphism from $\text{type}(C_{P_1})$ to $\text{type}(C_{P_2})$ that maps attribute ‘addr’ to attribute ‘address’. Then $\varphi^{-1}(\text{address.house}) = \text{addr.house}$ and $\varphi^{-1}(\text{address.street}) = \text{addr.street}$ exist, but $\varphi^{-1}(\text{address.city})$ does not exist. Let meth_1 be method ‘change_addr’ in class ‘Person1’ and meth_2 be method ‘move’ in class ‘Person2’. The generalisation of method ‘move’ in class ‘Person1’ is given by:

$$\begin{aligned} \varphi^{-1}(\text{meth}) = \\ \text{move } (h:\text{integer}, s:\text{string}) = \text{addr.house} := h; \text{addr.street} := s. \end{aligned}$$

\square

Now, we can define subclass morphisms based on subtype morphisms and generalisations of functional forms.

Definition 6.14 Let H be a well-defined class hierarchy, C_1 and C_2 be classes in H , φ be a type morphism from $\text{type}(C_1)$ to $\text{type}(C_2)$, and ψ be a function from $\text{funcs}(C_1)$ to $\text{funcs}(C_2)$, where

$$\text{funcs}(C) = \{\text{func}(C, \text{meth}) \mid \text{meth} \in \text{meths}(C)\}.$$

The pair (φ, ψ) is a subclass morphism from C_1 to C_2 if and only if:

1. φ is a subtype morphism
2. $\forall f \in \text{funcs}(C_1) [\psi(f) \leq_{\text{FUNC}}^w \mathcal{S}_\varphi(f)]$.

\square

There is an alternative, slightly different, definition of subclass morphisms based on subtype morphisms and generalisations of methods.

Definition 6.15 Let H be a well-defined class hierarchy, C_1 and C_2 be classes in H , φ be a type morphism from $\text{type}(C_1)$ to $\text{type}(C_2)$, and ψ be a function from $\text{meths}(C_1)$ to $\text{meths}(C_2)$. The pair (φ, ψ) is a subclass morphism from C_1 to C_2 if and only if:

1. φ is a subtype morphism
2. $\forall \text{meth} \in \text{meths}(C_1) [\text{func}(C_1, \varphi^{-1}(\psi(\text{meth}))) =_{\text{FUNC}}^w \text{func}(C_1, \text{meth})]$.

\square

To eliminate the difference between the two definitions of subclass morphism, we introduce specialised methods. A method is a specialised method if its functional form is a subfunction of the functional form of its generalisation.

Definition 6.16 Let H be a well-defined class hierarchy, C_1 and C_2 be classes in H , φ be a subtype morphism from $type(C_1)$ to $type(C_2)$, and $meth$ be a method in $meths(C_2)$. Then $meth$ is a specialised method according to φ , denoted by $spec(meth, \varphi)$, if and only if:

$$func(C_2, meth) \leq_{FUNC}^w \mathcal{S}_\varphi(func(C_1, \varphi^{-1}(meth))).$$

□

The difference between the two definitions of subclass morphism is eliminated if we require that all methods are specialised methods.

Lemma 6.17 Let H be a well-defined class hierarchy, C_1 and C_2 be classes in H , φ be a type morphism from $type(C_1)$ to $type(C_2)$, and ψ be a function from $meths(C_1)$ to $meths(C_2)$. Suppose:

$$\forall meth \in meths(C_1) [spec(\psi(meth), \varphi)].$$

Then:

$$\begin{aligned} \forall meth \in meths(C_1) [func(C_2, \psi(meth)) \leq_{FUNC}^w \mathcal{S}_\varphi(func(C_1, meth))] &\Leftrightarrow \\ \forall meth \in meths(C_1) [func(C_1, \varphi^{-1}(\psi(meth))) =_{FUNC}^w func(C_1, meth)] & \end{aligned}$$

Proof. \Rightarrow follows from the assumption, Definition 5.7, and Definition 6.11; \Leftarrow follows directly from the assumption. □

Finally, we extend the synthetic subclass relation with attribute renaming.

Definition 6.18 Let H be a well-defined class hierarchy and let C_1 and C_2 be classes in H . Then C_1 is a weak subclass of C_2 , denoted by $C_1 \preceq_{CLASS}^w C_2$, if and only if there is a subclass morphism from C_2 to C_1 . □

Note that a subclass morphism goes from superclass to subclass. The weak subclass relation is implied by the original subclass relation.

Lemma 6.19 Let H be a well-defined class hierarchy and let C_1 and C_2 be classes in H . Then:

$$C_1 \preceq_{CLASS} C_2 \Rightarrow C_1 \preceq_{CLASS}^w C_2.$$

Proof. The lemma follows from Definition 6.1, Lemma 6.9, and the fact that the subtype morphism of Lemma 6.9 preserves attribute names. □

The weak subclass relation is a pre-order.

Lemma 6.20 The weak subclass relation is reflexive and transitive.

Proof. To proof reflexivity, let C be a class. The pair (φ, ψ) , where φ is the identity homomorphism from $estruc(type(C))$ to $estruc(type(C))$ and ψ is the identity function on $funcs(C)$, is a subclass morphism. Hence, $C \preceq_{CLASS}^w C$. To proof transitivity, let C_1 , C_2 , and C_3 be classes, such that $C_1 \preceq_{CLASS}^w C_2$ and $C_2 \preceq_{CLASS}^w C_3$. Then there exists a subclass morphism (φ_1, ψ_1) from C_2 to C_1 and a subclass morphism from (φ_2, ψ_2) from C_3 to C_2 . But then $(\varphi_1 \circ \varphi_2, \psi_1 \circ \psi_2)$ is a subclass morphism from C_3 to C_1 , because function composition preserves the defining properties of subtype morphisms. Hence, $C_1 \preceq_{CLASS}^w C_3$. \square

The weak subclass relation is decidable, but has high complexity [27].

Theorem 6.21 The weak subclass relation is NP-complete.

Proof. The theorem is proven in Appendix A. \square

We conclude this chapter with an example.

Example 6.22 The following well-defined class hierarchy is a part of the definition of drawing tool A:

```

Class Square
Attributes
  x_left_up:integer
  y_left_up:integer
  width:integer
Methods
  set (x:integer, y:integer) =
    x_left_up := x; y_left_up := y
  translate (delta_x:integer, delta_y:integer) =
    x_left_up := x_left_up + delta_x;
    y_left_up := y_left_up + delta_y
Endclass.
```

The following well-defined class hierarchy is a part of the definition of drawing tool B:

```

Class Rectangle
Attributes
  x_left_up:integer
  y_left_up:integer
  width_x:integer
  width_y:integer
Methods
  set (x:integer, y:integer) =
    x_left_up := x; y_left_up := y
```

```

translate (delta_x:integer, delta_y:integer) =
  x_left_up := x_left_up + delta_x;
  y_left_up := y_left_up + delta_y
rotate = y_left_up := y_left_up + width_x;
width_x := width_y - width_x;
width_y := width_y - width_x;
width_x := width_x + width_y

```

Endclass.

The designer has chosen to model squares by the coordinates of the left upper corner and the width, and rectangles by the coordinates of the left upper corner and the width in both directions. Let C_S be class 'Square' and C_R be class 'Rectangle'. According to the synthetic subclass relation we have that $C_R \preceq_{CLASS}^w C_S$. This does not mean that every rectangle is a square; it only means that every description of a rectangle, as given by the designer, can be regarded as a description of a square (viz., by neglecting the width in one of the two directions). The fact that, in the real world, not every rectangle is a square indicates that the descriptions of squares and rectangles are not complete. Indeed, if we add a method to 'Square' that computes the area of a square and we add a method to 'Rectangle' that computes the area of a rectangle, then we no longer have that $C_R \preceq_{CLASS}^w C_S$, because the methods are different. \square

Chapter 7

Schema integration

In this chapter, we want to integrate schemas by their real world semantics. The attributes, formalised by types, describe the states of real world objects and the update methods, formalised by functions, describe the transitions between states. It follows that we can integrate classes by integrating types and functions separately. To integrate classes, we combine the join operator for underlying types (w.r.t. the subtype relation) and the join operator for functional forms (w.r.t. the subfunction relation) into a join operator for classes (w.r.t. the subclass relation).

More precisely, the join of class C_1 and class C_2 denoted by $C_1 \sqcup C_2$, is given by class C , such that:

1. $type(C)$ is the join of $type(C_1)$ and $type(C_2)$ w.r.t. \preceq_{TYPE}
2. $funcs(C) = \{f \mid \exists f_1 \in funcs(C_1) \exists f_2 \in funcs(C_2) [gen(f_1, type(C)) =_{FUNC}^w f =_{FUNC}^w gen(f_2, type(C))]\}$.

In the remainder of this chapter, we adapt this join operator to a join operator w.r.t. the weak subclass relation of the previous section. For that purpose, we define integration mappings, attribute joins, and method joins. Furthermore, we show that the integration operator w.r.t. the weak subclass relation can be used to integrate class hierarchies.

7.1 Integration of attributes

In this section, attribute joins are defined in terms of type joins according to an integration mapping, i.e., a set of attribute mappings. An attribute mapping is a partial function between attribute sets.

Definition 7.1 Let $H = \{C_1, \dots, C_n\}$ be a well-defined class hierarchy and, for every $i \in I = \{1, \dots, n\}$, let c_i be $name(C_i)$, and A_i be $atts(C_i)$. Furthermore,

let $\Phi = \{\varphi_{i,j} : A_i \rightarrow A_j\}_{i,j \in I}$ be a set of partial, injective functions, one for every $i, j \in I$. Then Φ is an integration mapping for H if and only if

1. $\forall i \in I [\varphi_{i,i} = id_{A_i}]$
2. $\forall i, j, k \in I \forall x \in dom(\varphi_{i,j}) \forall y \in dom(\varphi_{j,k}) \forall z \in range(\varphi_{j,k})$
 $[(\varphi_{i,j}(x) = y \wedge \varphi_{j,k}(y) = z) \Rightarrow \varphi_{i,k}(x) = z],$

where $dom(f)$ denotes the domain of a function f and $range(f)$ denotes the range of a function f . The set of all integration mappings for H is denoted by $IMaps_H$. \square

An integration mapping for class hierarchy H can be regarded as an integration description, describing for every class $C_1 \in H$ and every class $C_2 \in H$, how the attributes of C_1 correspond to the attributes of class C_2 . The join of a number of attribute sets according to an integration mapping is obtained by renaming the corresponding attributes and joining the corresponding types.

Definition 7.2 Let $CN(H)$ be the set of class names in H , $\nu : \wp(CN(H)) \rightarrow CN$ be an injective name-giving function for joins of classes, such that $\nu(\{c\}) = c$ for every $c \in CN(H)$, $AN(H)$ be the set of attribute names in H , and $\alpha : \wp(CN(H) \times AN(H)) \rightarrow AN$ be an injective name-giving function for joins of attributes, such that $\alpha(\{(c, a)\}) = a$ for every $(c, a) \in CN(H) \times AN(H)$. Furthermore, let $\{C_{i_1}, \dots, C_{i_p}\}$ be a subset of H and Φ be an integration mapping for H . The join of $\{A_{i_1}, \dots, A_{i_p}\}$ according to Φ is defined as follows:

$$\begin{aligned} \sqcup_{\Phi}(\{A_{i_1}, \dots, A_{i_p}\}) = & \{\alpha(\{(c_{i_1}, a_1), \dots, (c_{i_p}, a_p)\}) : T_1 \sqcup \dots \sqcup T_p \mid \\ & \varphi_{(i_1, i_2)}(a_1 : T_1) = a_2 : T_2 \wedge \\ & \vdots \\ & \varphi_{(i_{p-1}, i_p)}(a_{p-1} : T_{p-1}) = a_p : T_p \wedge \\ & T_1 \sqcup \dots \sqcup T_p \neq \perp\} \end{aligned}$$

where the type joins are given by (cf. [13]):

1. $T, T' \in CN$: $T \sqcup T' = \nu(\{c_i\}_{i \in J \cup J'})$ if $T = \nu(\{c_i\}_{i \in J})$ and $T' = \nu(\{c_i\}_{i \in J'})$
2. $T, T' \in \{\text{integer, rational, string}\}$: $T \sqcup T' = T$ if $T = T'$, and $T \sqcup T' = \perp$ otherwise
3. $T = \{U\}, T' = \{U'\}$: $T \sqcup T' = \{U \sqcup U'\}$ if $U \sqcup U' \neq \perp$, and $T \sqcup T' = \perp$ otherwise
4. $T = \langle l_i : U_i \mid i \in J \rangle, T' = \langle l_i : U'_i \mid i \in J' \rangle$:
 $T \sqcup T' = \langle l_i : U_i \sqcup U'_i \mid i \in J \cap J' \wedge U_i \sqcup U'_i \neq \perp \rangle$ if $\exists i \in J \cap J' [U_i \sqcup U'_i \neq \perp]$,
and $T \sqcup T' = \perp$ otherwise
5. otherwise: $T \sqcup T' = \perp$.

□

Note that the attribute join is computable. Finally, the joined attribute hierarchy according to an integration mapping is obtained by joining the attribute sets for every pair of classes.

Definition 7.3 Let H be a well-defined class hierarchy as in Definition 7.1, \tilde{H} be the same class hierarchy with methods removed and Φ be an integration mapping for H . The pre-join hierarchy of H according to Φ , denoted by \tilde{H}_Φ , is defined by:

$$\tilde{H}_\Phi = \{\tilde{C}_{(i,j)}\}_{i,j \in I}.$$

where $\tilde{C}_{(i,j)} = (\nu(\{c_i, c_j\}), \emptyset, \sqcup_\Phi(\{A_i, A_j\}), \emptyset)$. □

Example 7.4 For $i \in I$:

$$\tilde{C}_{(i,i)} = (c_i, \emptyset, A_i, \emptyset)$$

because of the constraints on ν , α , and Φ . □

Lemma 7.5 Let H be a well-defined class hierarchy as in Definition 7.1 and Φ be an integration mapping for H . For every $i \in I$, and $j \in I$:

$$\tilde{C}_{(i,i)} \preceq_{CLASS}^w \tilde{C}_{(i,j)}.$$

Proof. The lemma follows from the fact that $\varphi_{i,j}$ and α induce a subtype morphism φ from $type(\tilde{C}_{(i,j)})$ to $type(\tilde{C}_{(i,i)}) =_D type(C_i)$ as follows: if $a_1 : T_1 \in A_i$, $a_2 : T_2 \in A_j$, $\varphi_{i,j}(a_1 : T_1) = a_2 : T_2$, and $T_1 \sqcup T_2 \neq \perp$, then φ maps the arrow labeled $\alpha(\{(c_i, a_1), (c_j, a_2)\})$ to the arrow labeled a_1 and tree $estruc(type(T_1 \sqcup T_2))$ to tree $estruc(type(T_1))$. □

7.2 Integration of methods

In this section, method joins and pre-joins of classes are defined. To define method joins, we have to define generalisations of method sets.

Definition 7.6 Let H be a class hierarchy as in Definition 7.1, C_i and C_j be classes in H , M_i be $meths(C_i)$, and M_j be $meths(C_j)$. Furthermore, let Φ be an integration mapping for H , $\tilde{C}_{(i,j)}$ be $(\nu(\{c_i, c_j\}), \emptyset, \sqcup_\Phi(\{A_i, A_j\}), \emptyset)$, $\psi_{i,j}$ be the subtype morphism from $type(\tilde{C}_{(i,j)})$ to $type(C_i)$ induced by $\varphi_{i,j}$ and α , and $\psi_{j,i}$ be the subtype morphism from $type(\tilde{C}_{(i,j)})$ to $type(C_j)$ induced by $\varphi_{j,i}$ and α . The generalisation of M_i in $\tilde{C}_{(i,j)}$, denoted by $\psi_{i,j}^{-1}(M_i)$ and the generalisation of M_j in $\tilde{C}_{(i,j)}$, denoted by $\psi_{j,i}^{-1}(M_j)$, are defined by:

$$\begin{aligned} \psi_{i,j}^{-1}(M_i) &= \{\psi_{i,j}^{-1}(meth) \mid meth \in M_i \wedge spec(meth, \psi_{i,j})\} \\ \psi_{j,i}^{-1}(M_j) &= \{\psi_{j,i}^{-1}(meth) \mid meth \in M_j \wedge spec(meth, \psi_{j,i})\}. \end{aligned}$$

□

The join of a pair of method sets according to an integration mapping is obtained by combining the generalisations of the method sets.

Definition 7.7 Let $CN(H)$ be the set of class names in H , $MN(H)$ be the set of method names in H , and $\mu : \wp(CN(H) \times MN(H)) \rightarrow CN(H) \times MN(H)$ be an injective choice function for joins of methods, such that $\mu(V) \in V$ for every $V \in CN(H) \times MN(H)$. Furthermore, let C_i and C_j be classes in H , M_i be $meths(C_i)$, M_j be $meths(C_j)$, $\psi_{i,j}$ and $\psi_{j,i}$ be as defined in the previous definition, and

$$\begin{aligned} C_{(i,j)} &= (\nu(\{c_i, c_j\}), \emptyset, \sqcup_{\Phi}(\{A_i, A_j\}), \psi_{i,j}^{-1}(M_i)) \\ C_{(j,i)} &= (\nu(\{c_i, c_j\}), \emptyset, \sqcup_{\Phi}(\{A_i, A_j\}), \psi_{j,i}^{-1}(M_j)). \end{aligned}$$

The join of M_i and M_j according to Φ , denoted by $M_i \sqcup_{\Phi} M_j$, is defined as follows:

$$\begin{aligned} M_i \sqcup_{\Phi} M_j &= \{meth \mid m_i(P_i) = E_i \in \psi_{i,j}^{-1}(M_i) \wedge m_j(P_j) = E_j \in \psi_{j,i}^{-1}(M_j) \wedge \\ &\quad func(C_{(i,j)}, m_i(P_i) = E_i) =_{FUNC}^w func(C_{(j,i)}, m_j(P_j) = E_j) \wedge \\ &\quad (\mu(\{c_i, m_i\}, \{c_j, m_j\}) = (c_i, m_i) \Rightarrow meth = m_i(P_i) = E_i) \wedge \\ &\quad (\mu(\{c_i, m_i\}, \{c_j, m_j\}) = (c_j, m_j) \Rightarrow meth = m_j(P_j) = E_j)\}. \end{aligned}$$

□

Lemma 7.8 The method join is computable.

Proof. The definition gives a procedure to construct method joins, because the generalisation of a method is computable, $spec$ is decidable, and $=_{FUNC}^w$ is decidable. □

Finally, we define the pre-join of a pair of classes according to an integration mapping in terms of the attribute join and the method join.

Definition 7.9 Let $H = \{C_1, \dots, C_n\}$ be a well-defined class hierarchy and, for every $i \in \{1, \dots, n\}$, let c_i be $name(C_i)$, A_i be $atts(C_i)$, and M_i be $meths(C_i)$. Furthermore, let Φ be an integration mapping for H . The join of C_i and C_j according to Φ , denoted by $C_i \sqcup_{\Phi} C_j$, is defined as follows:

$$C_i \sqcup_{\Phi} C_j = (\nu(\{c_i, c_j\}), \emptyset, \sqcup_{\Phi}(\{A_i, A_j\}), M_i \sqcup_{\Phi} M_j).$$

□

7.3 Integration of class hierarchies

In this section, pre-join hierarchies and join hierarchies are defined. A pre-join hierarchy is a possible combination of pre-joins, one pre-join for every pair of classes.

Definition 7.10 Let H be a well-defined class hierarchy. The set of pre-join hierarchies of H , denoted by $prejoins(H)$, is defined by:

$$prejoins(H) = \{H_\Phi \mid \Phi \in IMaps_H\}$$

where $H_\Phi = \{C_1 \sqcup_\Phi C_2 \mid C_1 \in H \wedge C_2 \in H\}$. \square

To define join hierarchies, we have to define a specialisation relation on pre-join hierarchies.

Definition 7.11 Let H be a well-defined class hierarchy. Furthermore, let H_1 and H_2 be hierarchies in $prejoins(H)$. Then H_1 is a specialisation of H_2 , denoted by $H_1 \preceq_H H_2$, if and only if:

$$\forall C_1 \in H \forall C_2 \in H \exists D_1 \in H_1 \exists D_2 \in H_2 \\ [name(D_1) = name(D_2) = \nu(\{name(C_1), name(C_2)\}) \wedge D_1 \preceq_{CLASS}^w D_2].$$

\square

A join hierarchy of a class hierarchy is a most specialised pre-join hierarchy.

Definition 7.12 Let H be a well-defined class hierarchy. The set of join hierarchies of H , denoted by $joins(H)$ is defined as:

$$joins(H) = \{H_1 \in prejoins(H) \mid \\ \forall H_2 \in prejoins(H) [H_2 \preceq_H H_1 \Rightarrow H_1 \preceq_H H_2]\}.$$

\square

Note that every join hierarchy is a possible combination of joins, one join for every pair of classes. Finally, the set of joins of a pair of classes is defined as the set of pre-joins that occur in the join hierarchies.

Definition 7.13 Let H be a well-defined class hierarchy. Furthermore, let C_1 and C_2 be classes in H . The set of joins of C_1 and C_2 , denoted by $joins(C_1, C_2)$ is given by:

$$joins(C_1, C_2) = \{C \in H' \mid H' \in joins(H) \wedge name(C) = \nu(\{c_1, c_2\})\}.$$

\square

Theorem 7.14 The set of join hierarchies is computable.

Proof. Since $IMaps_H$ is finite and, \tilde{H}_Φ is computable for any $\Phi \in IMaps_H$, $prejoins(H)$ is computable. Furthermore, \preceq_H is decidable, because \preceq_{CLASS}^w is decidable. Hence, $joins(H)$ is computable. \square

7.4 Application

Every element of $joins(C_1, C_2)$ defines a superclass that can be used to factorise and integrate C_1 and C_2 . However, two pairs of classes should never be factorised into superclasses in isolation, because, otherwise, it could be that there is no join hierarchy that contains both superclasses. Therefore, if a two superclasses are chosen, then there should be a join hierarchy that contains both superclasses.

Example 7.15 The following class hierarchy is an extension of the first class hierarchy of Example 6.22.

```

Class Square
Attributes
  x_left_up:integer
  y_left_up:integer
  width:integer
Methods
  set (x:integer, y:integer) =
    x_left_up := x; y_left_up := y
  translate (delta_x:integer, delta_y:integer) =
    x_left_up := x_left_up + delta_x;
    y_left_up := y_left_up + delta_y
Endclass
Class SFigure
Attributes
  name:string
  first:Square
  second:Square
Methods
  copy1 = first := second
  copy2 = second := first
Endclass.

```

The following class hierarchy is an extension of the second class hierarchy of Example 6.22:

```

Class Rectangle
Attributes
  x_left_up:integer
  y_left_up:integer
  width_x:integer
  width_y:integer
Methods
  set (x:integer, y:integer) =

```

```

    x_left_up := x; y_left_up := y
    translate (delta_x:integer, delta_y:integer) =
      x_left_up := x_left_up + delta_x;
      y_left_up := y_left_up + delta_y
    rotate = y_left_up := y_left_up + width_x;
      width_x := width_y - width_x;
      width_y := width_y - width_x;
      width_x := width_x + width_y

```

Endclass

Class RFigure

Attributes

```

    code:integer
    left:Rectangle
    right:Rectangle

```

Methods

```

    r_copy = right := left
    l_copy = left := right

```

Endclass.

Let C_S be class 'Square' and C_R be class 'Rectangle'. If C is a class in $joins(C_S, C_R)$, then C belongs to \tilde{H}_Φ for some $\tilde{H}_\Phi \in joins(H)$ and, hence, C is given by an injective mapping from $atts(C_S)$ to $atts(C_R)$ that maps 'x_left_up' to 'x_left_up' or 'y_left_up', 'y_left_up' to 'y_left_up' or 'x_left_up', and 'width' to 'width_x' or 'width_y' (other mappings lead to join classes with fewer methods). The designer will probably choose to map 'x_left_up' to 'x_left_up' and 'y_left_up' to 'y_left_up'. In that case, there are two possibilities to factorise 'Square' and 'Rectangle'. If 'width' is mapped to 'width_x', then the join of 'Square' and 'Rectangle' is 'Square' itself and 'Rectangle' can be factorised as follows:

Class Rectangle Isa Square

Attributes

```

    width_y:integer

```

Methods

```

    rotate = y_left_up := y_left_up + width;
      width := width_y - width;
      width_y := width_y - width;
      width := width + width_y

```

Endclass.

Note that attribute 'width_x' has been renamed to 'width'. If 'width' is mapped to 'width_y', then the join of 'Square' and 'Rectangle' is 'Square' again and 'Rectangle' can be factorised as follows:

Class Rectangle Isa Square

Attributes

width_x:integer

Methods

rotate = y_left_up := y_left_up + width_x;

width_x := width - width_x;

width := width - width_x;

width_x := width_x + width

Endclass.

Let C_{SF} be class 'SFigure' and C_{RF} be class 'RFigure'. If C is a class in $joins(C_{SF}, C_{RF})$, then C is given by an injective mapping from $atts(C_{SF})$ to $atts(C_{RF})$ that maps 'first' to 'left' or 'right', and 'second' to 'left' or 'right' (other mappings lead to join classes with fewer methods). For both mappings, the join of 'SFigure' and 'Rfigure' is the same. So there is (up to attribute renaming) only one possibility to factorise 'SFigure' and 'RFigure':

Class SFigure Isa Figure**Attributes**

name:string

Endclass**Class RFigure Isa Figure****Attributes**

code:integer

first:Rectangle

second:Rectangle

Endclass**Class Figure****Attributes**

first:Square

second:Square

Methods

copy1 = first:=second

copy2 = second:=first

Endclass.

where 'Figure' is the join of 'SFigure' and 'RFigure'. \square

In the next part, we will extend our approach with schema transformations and syntactic properties of methods.

Part II

Chapter 8

Extended database schemas

In this part, we extend the approach developed in Part I using transformations on classes and syntactic properties of methods. We consider classes with attributes, constraints, update methods, and query methods. Recall that attributes correspond to structural and update methods to behavioural properties of real world objects. The constraints of a class instance correspond to structural properties of real world objects that are the same for all objects. The query methods of a class correspond to structural properties of real world objects that can be derived from other structural properties. The distinction between attributes and query methods that define derived attributes is useful, because equivalence of ordinary attributes is decidable, whereas equivalence of derived attributes is not.

Recall that our formalisation of attributes and update methods is similar to the approach of TM/FM [4]. We repeat this for constraints and query methods. Every constraint is formalised as a logical formula, the interpretation of which is a function from the powerset of the set of instances to the domain of the booleans, and every query method is formalised as a lambda expression, the interpretation of which is a function from the set of instances and the domains of the parameters to the corresponding codomain.

In the remainder of this chapter, we extend class hierarchies with simple static constraints (viz., keys), relax the inheritance mechanism for attributes, and extend the formalisation in terms of underlying types, underlying constraints, and functional forms.

8.1 Syntax

In this section, we introduce extended database schemas, consisting of classes with attributes, keys, update methods, and query methods.

A key consists of a sequence of attribute names and attribute selections. It

defines a constraint on the extensions of a class, saying that objects that have the same values for the key attributes must be the same ([62]). A query method has a name, a possibly empty list of parameters, a result, and a body, which consists of assignments to the result. An assignment can be a simple assignment, a method call, or the creation of an object. The result of a method can be used by other methods.

Definition 8.1 Extended class hierarchies are the sentences of the following BNF-grammar:

Hierarchy	::=	Class ⁺
Class	::=	'Class' CN ['Isa' CN ⁺] ['Attributes' Att ⁺] ['Constraints' Key ⁺] ['Methods' Meth ⁺] 'Endclass'
Att	::=	AN ':' Type
Type	::=	BasicType SetType RecordType CN
BasicType	::=	'integer' 'rational' 'string'
SetType	::=	'{' Type '}'
RecordType	::=	'<' FieldList '>'
FieldList	::=	Field Field ',' FieldList
Field	::=	L ':' Type
Key	::=	'key' Dest ⁺
Dest	::=	AN Dest ':' L
Meth	::=	UpMeth QueMeth
UpMeth	::=	MN '(' [ParList] ')' '=' UpAsnList
ParList	::=	Par Par ',' ParList
Par	::=	L ':' BasicType
UpAsnList	::=	UpAssign UpAssign ';' UpAsnList
UpAssign	::=	UpDest ':' UpSource 'insert(' UpSource ',' UpDest ')'
UpDest	::=	Dest
UpSource	::=	Source Object ':' MN '(' [ActParList] ')'
Source	::=	Term Term '+' Source Term '-' Source Term '×' Source Term '÷' Source
Term	::=	BC 'self' Path
Path	::=	L AN Path ':' L Path ':' AN
Object	::=	'self' Path
ActParList	::=	Source Source ';' ActParList
QueMeth	::=	MN '(' [ParList] '→' Result ')' '=' QueAsnList
Result	::=	L ':' Type
QueAsnList	::=	QueAssign QueAssign ';' QueAsnList

```

QueAssign    ::=    QueDest ':' QueSource |
                    'insert(' QueSource ',' QueDest ')'
QueDest      ::=    L | L '.' QueDest
QueSource    ::=    Source | 'new(' CN ',' NewParList ')'
NewParList   ::=    NewPar | NewPar ',' NewParList
NewPar       ::=    AN '=' UpSource | AN '= nself'

```

□

A class hierarchy is well-defined if it satisfies four conditions. The first condition is that class names are unique within the hierarchy, and classes only refer to classes in the hierarchy, and the **Isa** relation is acyclic. The second is that attribute names are unique within their class, attributes are well-typed, and inherited attributes are a specialisation of the corresponding attribute in the superclass. The third is that keys are well-defined (see section on underlying constraints). The fourth is that method names are unique within their class, methods are well-typed, methods are not recursive (direct or indirect), and inherited methods are a specialisation of the corresponding method in the superclass (see section on inheritance of methods).

8.2 Inheritance of attributes

In this subsection, we describe how attributes are inherited.

Example 8.2 The following class hierarchy introduces a class 'Person' and a class 'Employee', which specialises inherited attribute 'holiday_address' and adds attributes 'company' and 'salary':

```

Class Person
Attributes
  name : string
  mother : Person
  address : SimpleAddress
  holiday_address : SimpleAddress
Endclass
Class Employee Isa Person
Attributes
  holiday_address : Address
  company : string
  salary : integer
Endclass.

```

□

In definitions, every class is abbreviated to a 5-tuple $C = (c, S, A, K, M)$, where c is the name of the class, S is the set of names of its superclasses, A is the set of its new attributes, K is the set of its new keys, and M is the set of its new methods. The set of names of the superclasses of (abbreviated) class C is denoted by $sup_names(C)$. Furthermore, every class hierarchy is abbreviated to a set that contains the abbreviations of the classes in the hierarchy.

In general, object-oriented data models allow multiple inheritance, which means that a class can inherit attributes from more than one superclass. The mechanism defining how attributes are inherited imposes restrictions on multiple inheritance. If the inheritance mechanism is simple, as is the case in Chapter 3, then the restriction on multiple inheritance is severe.

Example 8.3 Let H be an abbreviated class hierarchy and $C = (c, S, A, K, M)$ be a class in H . According to the inheritance mechanism of Chapter 3, class ‘Assistant’ in the following class hierarchy has two different attributes with name ‘address’:

```

Class Student
Attributes
  name : string
  address : <house:integer,street:string,city:string>
Endclass
Class Employee
Attributes
  name : string
  address : <house:integer,zip:integer>
Endclass
Class Assistant Isa Student Employee
Endclass.
```

Since we require that attribute names must be unique, the class hierarchy is not well-defined. \square

If we use the inheritance mechanism of Chapter 3, then a necessary condition for class hierarchies to be well-defined is that inherited attributes with the same name must have the same type or must be redefined as a specialisation. Since we want to relax this restriction, we introduce a more complex inheritance mechanism that uses a meet operator on types to combine the different types in the superclasses.

Definition 8.4 Let H be a class hierarchy that satisfies the first condition for well-defined class hierarchies and $C = (c, S, A, K, M)$ be a class in H . The set of all attributes of C , denoted by $atts(C)$, is defined as:

$$atts(C) = A \cup \{a : T \mid inherits(a) \wedge T = \sqcap \{T' \mid inherits(a, T')\} \wedge \forall a' : T' \in A[a \neq a']\},$$

where

$$\begin{aligned} \text{inherits}(a, T') &= \\ &\exists C' \in H [(name(C), name(C')) \in isa(H) \wedge a:T' \in atts(C')] \\ \text{inherits}(a) &= \\ &\exists C' \in H \exists a':T' \in atts(C') [(name(C), name(C')) \in isa(H) \wedge a = a'] \end{aligned}$$

and meet operator \sqcap is defined as follows (cf. [13]):

1. $T, T' \in CN, T = name(D_1), T' = name(D_2)$:
 - (a) $T \sqcap T' = T$ if $T = T'$,
 - (b) $T \sqcap T'$ is a new class name corresponding to class:

$$\begin{aligned} &(d, \emptyset, \{a_1 : T_1 \in atts(D_1) \mid \forall a_2 : T_2 \in atts(D_2)[a_1 \neq a_2]\} \\ &\cup \{a_2 : T_2 \in atts(D_2) \mid \forall a_1 : T_1 \in atts(D_1)[a_1 \neq a_2]\} \\ &\cup \{b : U \mid \exists a_1 : T_1 \in atts(D_1) \exists a_2 : T_2 \in atts(D_2) \\ &\quad [b = a_1 = a_2 \wedge U = T_1 \sqcap T_2]\}, \emptyset), \\ &\text{where } T_1 \sqcap T_2 = d \text{ if } T_1 = T \wedge T_2 = T' \text{ or } T_1 = T' \wedge T_2 = T, \\ &\text{if } \forall a_1 : T_1 \in atts(D_1) \forall a_2 : T_2 \in atts(D_2)[a_1 = a_2 \Rightarrow T_1 \sqcap T_2 \neq \perp] \end{aligned}$$
 - (c) $T_1 \sqcap T_2 = \perp$ otherwise
2. $T, T' \in \{\text{integer, rational, string}\}$: $T \sqcap T' = T$ if $T = T'$ and $T \sqcap T' = \perp$ otherwise
3. $T = \{U\}, T' = \{U'\}$: $T \sqcap T' = \{U \sqcap U'\}$ if $U \sqcap U' \neq \perp$ and $T \sqcap T' = \perp$ otherwise
4. $T = \langle l_i : U_i \mid i \in I \rangle, T' = \langle l_i : U'_i \mid i \in I' \rangle$:
 - (a) $T \sqcap T' = \langle l_i : U \mid$

$$\begin{aligned} &(i \in I - I' \wedge U = U_i) \vee \\ &(i \in I' - I \wedge U = U'_i) \vee \\ &(i \in I \cap I' \wedge U = U_i \sqcap U'_i) \rangle \\ &\text{if } \forall i \in I \cap I' [U_i \sqcap U'_i \neq \perp] \end{aligned}$$
 - (b) $T \sqcap T' = \perp$ otherwise
5. otherwise: $T \sqcap T' = \perp$.

Since \sqcap is commutative and associative (modulo class renaming), $\sqcap\{T_1, \dots, T_n\} = T_1 \sqcap \dots \sqcap T_n$ is well-defined. \square

It follows that, in the case of single inheritance, an attribute that is well-typed in class C is also well-typed in every subclass of C . Since we will use the inheritance

mechanism of Definition 8.4 and we do not allow general redefinition of attributes, a necessary condition for class hierarchies to be well-defined is that inherited attributes with the same name must have a meet, because $\perp \notin WTypes$.

Example 8.5 Let H be the class hierarchy of Example 8.3. Let C_S be class ‘Student’, C_E be class ‘Employee’, and C_A be class ‘Assistant’. According to the inheritance mechanism of Definition 8.4, the attributes of class ‘Student’, class ‘Employee’, and class ‘Assistant’ are given by:

$$\begin{aligned}atts(C_S) &= \{\text{name:string, address:<house:integer, street:string, city:string>}\} \\atts(C_E) &= \{\text{name:string, address:<house:integer, zip:integer>}\} \\atts(C_A) &= \{\text{name:string, address:<house:integer, street:string, city:string, zip:integer>}\}.\end{aligned}$$

Hence, attributes have a unique name within their class and are well-typed. It follows that H is well-defined. \square

Every class in an extended class hierarchy has an underlying type. The definition of underlying types is the same as Definition 3.5 in Chapter 3.

8.3 Underlying constraints

In this subsection, we define underlying constraints of classes and class extensions.

Example 8.6 The following class hierarchy introduces a class ‘Person’ with key ‘name’:

```
Class Person
Attributes
  name : string
  mother : Person
  address : SimpleAddress
  holiday_address : SimpleAddress
Constraints
  key name
Endclass.
```

\square

The set of all keys of a class consists of both the new and inherited keys.

Definition 8.7 Let H be a class hierarchy that satisfies the first condition for well-defined class hierarchies and $C = (c, S, A, K, M)$ be a class in H . The set of all keys of C is defined as:

$$keys(C) = K \cup \{r \mid \exists C' \in H [(name(C), name(C')) \in isa(H) \wedge p \in keys(C')]\}.$$

A key **key** $p_1 \cdots p_n$ in $keys(C)$ is well-defined if every p_i is the labeling of a path in $struc(type(C))$, starting at the root. \square

It follows that a key that is well-defined in class C is also well-defined in every subclass of C .

Every class in an extended class hierarchy has an underlying constraint, i.e., a predicate that has to be satisfied by the extensions of the class. More precisely, an extension of a class is a set of objects of its underlying type that satisfies the underlying constraint. An underlying constraint is a constraint on objects of the same class, not on objects of different classes. Constraints on objects of different classes can be combined into an underlying constraint for the class hierarchy, i.e., a predicate that has to be satisfied by the extensions of the class hierarchy. More precisely, an extension of a class hierarchy is a combination of class extensions, one for every class in the hierarchy, that satisfies the underlying constraint of the class hierarchy. However, extensions of class hierarchies are not considered in this thesis. For more details, see [4] and [5].

The underlying constraint of a class is the conjunction of identifier uniqueness, partial referential integrity, i.e., referential integrity within the same class, and key uniqueness.

Definition 8.8 Let H be a class hierarchy that satisfies the first and second condition for well-defined class hierarchies and C be a class in H . The set of extensions of class C is defined as:

$$exts(C) = \{e \subseteq ext(type(C)) \mid constraint(C, e)\},$$

where $constraint(C, e)$ is defined as the conjunction of

1. identifier uniqueness: $\forall x \in e \forall y \in e [x.id = y.id \Rightarrow x \cong_{TERM} y]$
2. partial referential integrity: a conjunction of formulas, one formula $\forall x \in e [refint(x.l_1, l_2, \dots, l_n, 0)]$ for every path $l_1 \cdots l_n name(C)$ in $paths(type(C))$
3. key uniqueness: one formula for every key **key** $p_1 \cdots p_n$ in $keys(C)$:
 $\forall x \in e \forall y \in e [(x.p_1 \cong_{TERM} y.p_1 \wedge \cdots \wedge x.p_n \cong_{TERM} y.p_n) \Rightarrow x \cong_{TERM} y]$

and

$$\begin{aligned} paths(t) &= \{t\} \text{ if } t \in CN \\ paths(B) &= \{B\} \text{ if } B \in BTypes \\ paths(\{v\}) &= \{\in.p \mid p \in paths(v)\} \\ paths(< l_1 : v_1, \dots, l_n : v_n >) &= \\ &\quad \{l_1.p \mid p \in paths(v_1)\} \cup \cdots \cup \{l_n.p \mid p \in paths(v_n)\} \\ paths(\mu t.\alpha) &= paths(\alpha) \end{aligned}$$

and

$$\begin{aligned}
\text{refint}(p, \text{empty}, i) &= \exists y \in e [p \cong_{\text{TERM}} y] \\
\text{refint}(p, l, i) &= \exists y \in e [p.l \cong_{\text{TERM}} y] \\
\text{refint}(p, \in, i) &= \forall x_{i+1} \in p \exists y \in e [x_{i+1} \cong_{\text{TERM}} y] \\
\text{refint}(p, l.q, i) &= \text{refint}(p.l, q, i) \\
\text{refint}(p, \in.q, i) &= \forall x_{i+1} \in p \text{refint}(x_{i+1}, q, i+1).
\end{aligned}$$

□

Note that every occurrence of $\text{name}(C)$ in $\text{atts}(C)$ uniquely corresponds to a path $p.\text{name}(C)$ in $\text{paths}(\text{type}(C))$.

Example 8.9 Let C be the following class:

```

Class Person
Attributes
  name : string
  teachers : {<name:string,children:{Person}>}
Endclass.

```

The partial referential integrity formula for C is given by:

$$\begin{aligned}
\forall x \in e[\text{refint}(x.\text{teachers}, \in.\text{children}.\in, 0)] = \\
\forall x \in e[\forall x_1 \in x.\text{teachers} \forall x_2 \in x_1.\text{children} \exists y \in e[x_2 \cong_{\text{TERM}} y]].
\end{aligned}$$

□

8.4 Inheritance of methods

In this subsection, we describe how methods are inherited. Furthermore, we redefine well-typed methods, functional forms of methods, equality of functional forms, and the subfunction relation on functional forms.

Example 8.10 The following class hierarchy introduces a class ‘Person’ with an update method ‘stay’ and a query method ‘age’:

```

Class Person
Attributes
  name : string
  dob : <d:integer,m:string,y:integer>
  mother : Person
  address : SimpleAddress
  holiday_address : SimpleAddress
Methods
  stay () = holiday_address := address
  age (year:integer → result:integer) =
    result := year - dob.y
Endclass.

```

□

The set of all methods of a class consists of both the new and inherited, and both the update and query methods.

Definition 8.11 Let H be a class hierarchy that satisfies the first condition for well-defined class hierarchies and $C = (c, S, K, A, M)$ be a class in H . The set of all update methods of C , denoted by $u_meths(C)$, is defined as follows:

$$u_meths(C) = \{m(P)=E \mid m(P)=E \in M\} \cup \\ \{m(P)=E \mid \exists C' \in H [name(C') \in S \wedge m(P)=E \\ \in u_meths(C')] \wedge \forall m'(P')=E' \in M [m \neq m']\}.$$

The set of all query methods of C , denoted by $q_meths(C)$, is defined as follows:

$$q_meths(C) = \{m(P \rightarrow l:T)=E \mid m(P \rightarrow l:T)=E \in M\} \cup \\ \{m(P \rightarrow l:T)=E \mid \exists C' \in H [name(C') \in S \wedge \\ m(P \rightarrow l:T)=E \in meths(C')] \wedge \\ \forall m'(P' \rightarrow l':T')=E' \in M [m \neq m']\}.$$

Finally, $meths(C)$ is the union of $u_meths(C)$ and $q_meths(C)$. □

Since the methods are different from the methods of Chapter 3, we have to redefine well-typed methods.

Definition 8.12 Let H be a class hierarchy that satisfies the first and second condition for well-defined class hierarchies and C be a class in H . Furthermore, let $m(P \rightarrow l : T) = E$ be a query method in $q_meths(C)$. The query method is well-typed if for every assignment $d := s$ (resp., $\text{insert}(s, d)$) in E :

1. d starts with l (i.e., only assignments to the result of the method)
2. if $\text{new}(c', a_1 = e_1, \dots, a_n = e_n)$ occurs in s , such that c' is the name of class C' in H , then:
 - (a) $(name(C), c')$ is an element of $sub(H)$
(i.e., only creation of objects in class C and superclasses of C)
 - (b) C' has exactly n attributes: $a_1 : T_1, \dots, a_n : T_n$
 - (c) for every $i \in \{1, \dots, n\}$, the type of e_i is a subtype of T_i according to subtype relation \leq_H
3. if $p.m'(e'_1, \dots, e'_n)$ occurs in s , such that p has type c' , where c' is the name of class C' in H , and m' is the name of method $meth$ in C' , then:
 - (a) $meth$ has exactly n parameters: $p_1 : T_1, \dots, p_n : T_n$
 - (b) for every $i \in \{1, \dots, n\}$, the type of e'_i is a subtype of T_i according to subtype relation \leq_H

4. the type of s is a subtype of the type of d according to subtype relation \leq_H (resp., the type of d is $\{T\}$ and the type of s is a subtype of T),

where the type of a source or a destination is defined as follows:

1. the type of an integer constant is integer
2. the type of a rational constant is rational
3. the type of a string constant is string
4. the type of l is B if $l:B$ is a parameter in P
5. the type of **self** is $name(C)$
6. the type of $a.l_1 \dots l_n$ is $T.l_1 \dots l_n$ if $a:T$ is an attribute in $atts(C)$ and $T.l_1 \dots l_n \neq \perp$
7. the type of a complex term follows from the types of the subterms and the standard rules for $+$ (addition for integers and rationals; concatenation for strings), $-$ (subtraction for integers and rationals), \times (multiplication for integers and rationals), and \div (division for rationals only)
8. the type of **new**(d, e_1, \dots, e_n) is d
9. the type of a call to method $m'(P' \rightarrow l' : T') = E'$ is T'
10. otherwise, the type of a source or a destination is undefined

and

$$\begin{aligned}
 & \langle l_1 : T_1, \dots, l_n : T_n \rangle . l = T \text{ if } l \in \{l_1, \dots, l_n\} \wedge T = T_i \\
 & \langle l_1 : T_1, \dots, l_n : T_n \rangle . l = \perp \text{ if } l \notin \{l_1, \dots, l_n\} \\
 & c'.a = T \text{ if } \exists C' \in H \exists a' : T' \in atts(C') [c' = name(C') \wedge a = a' \wedge T = T'] \\
 & c'.a = \perp \text{ if } \forall C' \in H \forall a' : T' \in atts(C') [c' \neq name(C') \vee a \neq a'].
 \end{aligned}$$

Now, let $m(P) = E$ be an update method in $u_meths(C)$. The update method is well-typed if for every assignment $d := s$ in E , the type of s is a subtype of the type of d (according to \leq_H) and if for every method call $s.m'(v_1, \dots, v_n)$ in E , the following holds: the type of s must be $name(D)$ for some $D \in H$ and there must be a method $m'(P' \rightarrow l' : T') = E'$ in $meths(D)$, such that the type of each v_i is a subtype (according to \leq_H) of the type of the corresponding formal parameter in P' . The type of a source or destination in an update method is defined as for query methods, with the exception that the third item (regarding selections of the result of a query method) is removed. \square

Since we will use the inheritance mechanism of Definition 8.11 for methods, a necessary condition for class hierarchies to be well-defined is that inherited methods with the same name must be the same or must be redefined as a specialisation.

Example 8.13 According to the inheritance mechanism of Definition 8.11, class ‘Employee’ in the following class hierarchy has one method, viz., ‘stay() = holiday_address := address’:

```

Class Person
Attributes
  name : string
  mother : Person
  address : SimpleAddress
  holiday_address : SimpleAddress
Methods
  stay() = holiday_address := address
Endclass
Class Employee Isa Person
Attributes
  holiday_address : Address
  company : string
  salary : integer
Endclass.

```

Although method ‘stay’ is well-typed in class ‘Person’, it is not well-typed in class ‘Employee’, because attribute ‘holiday_address’ has been specialised (if we do not allow attribute specialisation, then every method that is well-typed in class C is also well-typed in every subclass of C). Hence, class ‘Employee’ is not well-defined. One way to repair this is to use a different inheritance mechanism for methods, where ill-typed methods are redefined. However, we do not allow general redefinition of methods. Nevertheless, class ‘Employee’ can be redefined as a well-defined class by specialising attribute ‘address’ in the same way as attribute ‘holiday_address’, i.e., adding attribute ‘address:Address’ to the new attributes of class ‘Employee’. For a more complicated example, let class ‘Person’ and class ‘AgedPerson’ be given by:

```

Class Person
Attributes
  name : string
  address : SimpleAddress
  mother : Person
  grandmother : Person
Methods
  define_grandmother () =
    grandmother := mother.mother1()
  mother1 ( $\rightarrow$  p:Person) =
    p := mother
Endclass

```

```

Class AgedPerson Isa Person
Attributes
  address : Address
  grandmother : AgedPerson
Endclass.

```

Method 'define_grandmother' is well-typed in class 'Person', but not in class 'AgedPerson', because attribute 'grandmother' has been specialised. Hence, class 'AgedPerson' is not well-defined. However, class 'AgedPerson' can be redefined as a well-defined class by specialising both attribute 'mother' and method 'mother1':

```

Class AgedPerson Isa Person
Attributes
  address : Address
  mother : AgedPerson
  grandmother : AgedPerson
Methods
  define_grandmother () =
    grandmother := mother.mother1()
  mother1 (→ p:AgedPerson) =
    p := mother
Endclass.

```

□

Every class in an extended class hierarchy has a set of functional forms (one for each of its methods). The definition of these functional forms is more complicated than Definition 3.12 in Chapter 3, because of the **new**-statement. The functional form of a query (resp., an update) method is a function of which the body is an accumulation of the assignments to the result (resp., to the input) of the method.

Definition 8.14 First, we postulate an injective function $newid:integer \rightarrow oid$ that maps every integer to a unique object identifier.

Let $H = \{C_1, \dots, C_n\}$ be a well-defined class hierarchy, $\{c_1, \dots, c_n\}$ be the set of names of the classes in H , C be a class in H , c be $name(C)$, and $\{a_1 : T_1, \dots, a_k : T_k\}$ be $atts(C)$. Furthermore, let $meth = m(P \rightarrow l : T) = E$ be a query method in $q_meths(C)$. The functional form of query method $meth$ in class C , denoted by $func_q(C, meth)$, is defined as:

$$func_q(C, meth) = \lambda count:integer \lambda obj:type(C) \lambda P. eval_q(E)(obj, \rho_0(T), count),$$

where

$$\rho_0(d) = \perp \text{ if } d \in CN,$$

$$\begin{aligned}
\rho_0(B) &= \perp \text{ if } B \in \{\text{integer, rational, string}\}, \\
\rho_0(\{U\}) &= \perp, \\
\rho_0(< l_1 : U_1, \dots, l_n : U_n >) &= < l_1 = \rho_0(U_1), \dots, l_n = \rho_0(U_n) >,
\end{aligned}$$

and the syntactic evaluation of body E in state $\sigma = (obj, r, N)$ (where obj is the object that is queried by the method, r is the current value of the result of the method, and N is the current number of objects), denoted by $eval_q(E)\sigma$, is given by:

$$\begin{aligned}
eval_q(L_1; L_2)\sigma &= eval_q(L_2)(eval_q(L_1)\sigma), \\
eval_q(l := s)(obj, r, N) &= \\
& (obj, \pi_1(ev(s)(obj, r, N)), \pi_2(ev(s)(obj, r, N))), \\
eval_q(l.l_1 \dots l_n := s)(obj, r, N) &= \\
& (obj, r[l_1 \dots l_n = \pi_1(ev(s)(obj, r, N))], \pi_2(ev(s)(obj, r, N))), \\
eval_q(\text{insert}(s, l))(obj, r, N) &= \\
& (obj, r \cup \{\pi_1(ev(s)(obj, r, N))\}, \pi_2(ev(s)(obj, r, N))), \\
eval_q(\text{insert}(s, l.l_1 \dots l_n))(obj, r, N) &= \\
& (obj, r[l_1 \dots l_n = (r.l_1 \dots l_n \cup \{\pi_1(ev(s)(obj, r, N))\})], \\
& \pi_2(ev(s)(obj, r, N))),
\end{aligned}$$

where

$$\begin{aligned}
ev(b)(obj, r, N) &= (b, N), \\
& \text{if } b \in BC \\
ev(\text{self})(obj, r, N) &= (obj, N), \\
ev(l')(obj, r, N) &= (l', N) \\
& \text{if } l' \in L \text{ and } l' \neq l, \\
ev(l)(obj, r, N) &= (r, N) \\
ev(l.l_1 \dots l_n)(obj, r, N) &= (r.l_1 \dots l_n, N) \\
ev(l_1 \dots l_n)(obj, r, N) &= (obj.l_1 \dots l_n, N) \\
& \text{if } l_1 \in AN, \\
ev(s_1 \theta s_2)(obj, r, N) &= (\pi_1(ev(s_1)(obj, r, N)) \theta \pi_1(ev(s_2)(obj, r, N)), f) \\
& \text{if } \theta \in \{+, -, \times, \div\} \\
ev(\text{new}(c, a_1 = e'_1, \dots, a_k = e'_k))(obj, r, N) &= \\
& (\mu y. < \text{id} = \text{newid}(N), a_1 = \pi_1(ev(e'_1)(obj, r, N)), \dots, \\
& a_k = \pi_1(ev(e'_k)(obj, r, N)) >, N + 1), \\
& \text{where } ev(\text{nself})(obj, r, N) = (y, N) \\
ev(p.m'(e'_1, \dots, e'_n))(obj, r, N) &= \\
& \pi_{2,3}(\text{func}_q(C', \text{meth}')(N)(\pi_1(ev(p)(obj, r, N))) \\
& (\pi_1(ev(e'_1)(obj, r, N))) \dots (\pi_1(ev(e'_n)(obj, r, N))))), \\
& \text{where } C' \text{ is the class in } H \text{ and } \text{meth}' \text{ is the method in } \text{meths}(C'), \\
& \text{such that } \text{name}(C') \text{ is the type of } p \text{ and } \text{meth}' \text{ has name } m'
\end{aligned}$$

and

$$\begin{aligned}
& \langle l_1 = v_1, \dots, l_n = v_n \rangle [l_1 = v] = \\
& \quad \langle l_1 = v, \dots, l_n = v_n \rangle \\
& \langle l_1 = v_1, \dots, l_n = v_n \rangle [l_1.l'_1 \dots l'_n = v] = \\
& \quad \langle l_1 = v_1[l'_1 \dots l'_n = v], \dots, l_n = v_n \rangle.
\end{aligned}$$

Now, let $meth = m(P) = E$ be an update method in $meths(C)$. The functional form of update method $meth$ in class C , denoted by $func_u(C, meth)$, is defined as:

$$\begin{aligned}
func_u(C, meth) &= \lambda count:integer \lambda obj:type(C) \lambda P.eval_u(E) \\
& \quad (\mu \delta(obj). \langle id = obj.id, a_1 = \sigma_0(a_1, T_1), \dots, a_k = \sigma_0(a_k, T_k) \rangle, count),
\end{aligned}$$

where

$$\begin{aligned}
\sigma_0(r, d) &= obj.r \text{ if } d \in CN, \\
\sigma_0(r, B) &= obj.r \text{ if } B \in \{\text{integer, rational, string}\}, \\
\sigma_0(r, \{U\}) &= obj.r, \\
\sigma_0(r, \langle l_1 : U_1, \dots, l_n : U_n \rangle) &= \langle l_1 = \sigma_0(r.l_1, U_1), \dots, l_n = \sigma_0(r.l_n, U_n) \rangle,
\end{aligned}$$

and the syntactic evaluation of body E in state $\sigma = (obj, N)$ (where obj is the object that is updated by the method and N is the current number of objects), denoted by $eval(E)\sigma$, is given by:

$$\begin{aligned}
eval_u(L_1; L_2)\sigma &= eval_u(L_2)(eval_u(L_1)\sigma), \\
eval_u(a_1 := s)(obj, N) &= \\
& \quad (\mu \delta(obj). (cut_{\delta(obj)}(\langle id = e_0, a_1 = \pi_1(ev(s)(obj, N)), \dots, a_k = e_k \rangle)), \\
& \quad \pi_2(ev(s)(obj, N))), \\
eval_u(a_1.l_1 \dots l_n := s)(obj, N) &= \\
& \quad (\mu \delta(obj). (cut_{\delta(obj)}(\langle id = e_0, a_1 = e.a_1[l_1 \dots l_n = \pi_1(ev(s)(obj, N))], \dots, \\
& \quad a_k = e_k \rangle)), \pi_2(ev(s)(obj, N))), \\
eval_u(insert(s, a_1))(obj, N) &= \\
& \quad (\mu \delta(obj). (cut_{\delta(obj)}(\langle id = e_0, a_1 = e.a_1 \cup \{\pi_1(ev(s)(obj, N))\}, \dots, \\
& \quad a_k = e_k \rangle)), \pi_2(ev(s)(obj, N))), \\
eval_u(insert(s, a_1.l_1 \dots l_n))(obj, N) &= \\
& \quad (\mu \delta(obj). (cut_{\delta(obj)}(\langle id = e_0, a_1 = e.a_1[l_1 \dots l_n = e.a_1.l_1 \dots l_n \cup \\
& \quad \{\pi_1(ev(s)(obj, N))\}], \dots, a_k = e_k \rangle)), \pi_2(ev(s)(obj, N))),
\end{aligned}$$

where

$$\begin{aligned}
ev(b)(obj, N) &= (b, N) \\
& \text{if } b \in BC \\
ev(\mathbf{self})(obj, N) &= (obj, N), \\
ev(l)(obj, N) &= (l, N) \\
& \text{if } l \in L, \\
ev(l_1 \dots l_n)(obj, N) &= (obj.l_1 \dots l_n, N) \\
& \text{if } l_1 \in AN,
\end{aligned}$$

$$\begin{aligned}
ev(s_1 \theta s_2)(obj, N) &= (\pi_1(ev(s_1)(obj, N)) \theta \pi_1(ev(s_2)(obj, N)), f) \\
&\text{if } \theta \in \{+, -, \times, \div\}, \\
ev(p.m'(e'_1, \dots, e'_n))(obj, r, N) &= \\
&\pi_{2,3}((func_q(C', meth'))(N)(\pi_1(ev(p)(obj, r, N))) \\
&\quad (\pi_1(ev(e'_1)(obj, r, N))) \cdots (\pi_1(ev(e'_n)(obj, r, N))))), \\
&\text{where } C' \text{ is the class in } H \text{ and } meth' \text{ is the method in } meths(C'), \\
&\text{such that } name(C') \text{ is the type of } p \text{ and } meth' \text{ has name } m'
\end{aligned}$$

and

$$\begin{aligned}
cut_x(x') &= x' \text{ if } x' \in Var \\
cut_x(b) &= b \text{ if } b \in Cons \\
cut_x(\{e_1, \dots, e_n\}) &= \{cut_x(e_1), \dots, cut_x(e_n)\} \\
cut_x(<l_1 = e_1, \dots, l_n = e_n>, V) &= <l_1 = cut_x(e_1), \dots, l_n = cut_x(e_n)> \\
cut_x(\mu x.e') &= x \\
cut_x(\mu x'.e') &= \mu x'.e' \text{ if } x' \neq x.
\end{aligned}$$

□

Example 8.15 Let H be the first class hierarchy of Example 8.11. Let C_P be class 'Person' and C_E be class 'Employee' after attribute 'address:Address' has been added. The functional forms of method 'stay' in class 'Person' and class 'Employee' are given by:

$$\begin{aligned}
&\lambda i:\text{integer} \lambda o:\text{type}(C_P) . \\
&\quad \mu \delta(o). <\text{id}=o.\text{id}, \text{name}=o.\text{name}, \text{mother}=o.\text{mother}, \\
&\quad \quad \text{address}=o.\text{address}, \text{holiday_address}=o.\text{address}>, \\
&\lambda i:\text{integer} \lambda o:\text{type}(C_E) . \\
&\quad \mu \delta(o). <\text{id}=o.\text{id}, \text{name}=o.\text{name}, \text{mother}=o.\text{mother}, \text{address}=o.\text{address}, \\
&\quad \quad \text{holiday_address}=o.\text{address}, \text{company}=o.\text{company}, \text{salary}=o.\text{salary}>.
\end{aligned}$$

□

Since the functional forms are different from the functional forms of Chapter 5, we have to redefine the equality relation and the subfunction relation on functional forms. Equality of functional forms is defined as extensional equality of their semantic counterparts.

Definition 8.16 Let τ and σ be types. Furthermore, let

$$\begin{aligned}
F &= \lambda count:\text{integer} \lambda obj:\tau \lambda p_1:\tau_1 \cdots \lambda p_k:\tau_k . f, \\
G &= \lambda count:\text{integer} \lambda obj:\tau \lambda q_1:v_1 \cdots \lambda q_m:v_m . g
\end{aligned}$$

be functions in $functions(\tau, \sigma)$. Equality of F and G , denoted by $F =_{FUNC} G$, is defined by:

$$F =_{FUNC} G \Leftrightarrow \forall i \in [\text{integer}] \forall e_0 \in [\tau] \forall e_1 \in [\tau_1] \cdots \forall e_k \in [\tau_k] [F_{\square}(\vec{e}) =_D G_{\square}(\vec{e})],$$

where $\vec{e} = (i, e_0, e_1, \dots, e_k)$. Note that equality requires that $k = m$ and $\forall j \in \{1, \dots, n\} [\tau_j =_D \nu_j]$. \square

Equality of functional forms can be weakened by permuting parameters and replacing object identifiers.

Definition 8.17 Let τ and σ be types. Let $F = \lambda count:integer \lambda obj:\tau \lambda P. f$ and $G = \lambda count:integer \lambda obj:\tau \lambda Q. g$ be functions in $functions(\tau, \sigma)$. Then F is weakly equal to G , denoted by $F =_{FUNC}^w G$, if and only if there is a permutation \tilde{Q} of Q and a bijective function h from $newids(F)$ to $newids(G)$, such that:

$$h(F) =_{FUNC} \lambda obj:\tau \lambda \tilde{Q}. g,$$

where $newids(F)$ is the set of expressions of the form $newid(e)$ occurring in F and $h(F)$ is obtained from F by replacing every occurrence of e in $newids(F)$ by $h(e)$. \square

To define the weak subfunction relation on functional forms, we have to introduce the generalisation of a functional form.

Definition 8.18 Let τ_1 and σ_1 be types, and $F = \lambda count:integer \lambda obj:\tau_1 \lambda P. e$ be a functional form in $functions(\tau_1, \sigma_1)$. Furthermore, let τ_2 and σ_2 be types, such that $\tau_1 \preceq_{TYPE} \tau_2$ and $\sigma_1 \preceq_{TYPE} \sigma_2$. The generalisation of F to a functional form in $functions(\tau_2, \sigma_2)$, denoted by $gen(F, \tau_2, \sigma_2)$, is defined as follows:

$$gen(F, \tau_2, \sigma_2) = \lambda count:integer \lambda obj:\tau_2 \lambda \tilde{P}. (proj(e, estruc(\sigma_2), obj, \emptyset)),$$

where \tilde{P} is obtained from P by removing parameters that do not occur in $proj(e, estruc(\sigma_2), obj, \emptyset)$ and $proj$ is the same as in Chapter 5, with the following additional item:

$$proj(newid(e), T, \bar{l}, V) = newid(e).$$

\square

The weak subfunction relation on functional forms is defined as weak equality of their generalisations.

Definition 8.19 Let τ_1 and σ_1 be types, and let F_1 be a functional form in $functions(\tau_1, \sigma_1)$. Furthermore, let τ_2 and σ_2 be types, such that $\tau_1 \preceq_{TYPE} \tau_2$ and $\sigma_1 \preceq_{TYPE} \sigma_2$, and let F_2 be a functional form in $functions(\tau_2, \sigma_2)$. Then F_1 is a weak subfunction of F_2 , denoted by $F_1 \leq_{FUNC}^w F_2$, if and only if:

$$gen(F_1, \tau_2, \sigma_2) =_{FUNC}^w F_2.$$

\square

Chapter 9

Schema transformations

In this chapter, we introduce type transformations to extend the basis for the comparison of attributes, constraints, and methods. We consider renaming and aggregation of fields in record types, adding fields to record types, and combinations of these operations. Renaming and aggregation are capacity preserving, i.e., they do not add data capacity to a type and its instances. Adding fields is capacity augmenting, i.e., it adds data capacity. The motivation for choosing these transformations is that renaming and aggregation together are complete w.r.t. data capacity and adding id-fields induces objectification, which is an important transformation for our schemas. we give an overview of type transformations and show how type transformations induce schema transformations.

Furthermore, we show that a type transformation induces a transformation on predicates and a transformation on functions, which can be combined into a transformation on classes. The type transformation defines the transformation for the attribute part of a class, the transformation on predicates for the constraint part, and the transformation on functions for the method part.

9.1 Type transformations

The set of basic type transformations consists of renaming, aggregation, and objectification operations (cf. [1] and [24]).

Definition 9.1 Renaming is defined as a function of type $\mathcal{L} \rightarrow (\mathcal{L} \rightarrow (Types \rightarrow Types))$:

$$\begin{aligned} rename(l_i)(l)(t) &= t \text{ if } t \in TypeVar \\ rename(l_i)(l)(B) &= B \text{ if } B \in BTypes \\ rename(l_i)(l)(\{v\}) &= \{v\} \\ rename(l_i)(l)(\langle l_1 : v_1, \dots, l_n : v_n \rangle) &= \langle l_1 : v_1, \dots, l_n : v_n \rangle \end{aligned}$$

$$\begin{aligned}
& \text{if } l_i \notin \{l_1, \dots, l_n\} \text{ or } l \in \{l_1, \dots, l_n\}, \\
& \text{rename}(l_i)(l)(\langle l_1 : v_1, \dots, l_n : v_n \rangle) = \langle l_1 : v_1, \dots, l : v_i, l_n : v_n \rangle \\
& \text{if } l_i \in \{l_1, \dots, l_n\} \text{ and } l \notin \{l_1, \dots, l_n\}, \\
& \text{rename}(l_i)(l)(\mu t. \alpha) = \mu t. (\text{rename}(l_i)(l)(\alpha)).
\end{aligned}$$

We distinguish between two kinds of aggregation: tupling and aggregation within a record type. Tupling is defined as a function of type $\mathcal{L} \rightarrow (\text{Types} \rightarrow \text{Types})$:

$$\text{tuple}(l)(\tau) = \langle l : \tau \rangle.$$

The inverse of tupling is de-tupling, defined as a function of type $\text{Types} \rightarrow \text{Types}$:

$$\text{de_tuple}(\langle l : \tau \rangle) = \tau.$$

For all other cases, we have: $\text{de_tuple}(v) = v$. Aggregation within a record type is defined as a function of type $\wp_{fin}(\mathcal{L}) \rightarrow (\mathcal{L} \rightarrow (\text{Types} \rightarrow \text{Types}))$:

$$\begin{aligned}
& \text{aggregate}(\{l_i, l_{i+1}, \dots, l_j\})(l)(\langle l_1 : v_1, \dots, l_n : v_n \rangle) = \\
& \quad \langle l_1 : v_1, \dots, l : \langle l_i : v_i, \dots, l_j : v_j \rangle, \dots, l_n : v_n \rangle \\
& \quad \text{if } \{l_i, l_{i+1}, \dots, l_j\} \subseteq \{l_1, \dots, l_n\} \text{ and } l \notin (\{l_1, \dots, l_n\} - \{l_i, l_{i+1}, \dots, l_j\}) \\
& \text{aggregate}(\{l_i, l_{i+1}, \dots, l_j\})(l)(\mu t. \alpha) = \\
& \quad \mu t. (\text{aggregate}(\{l_i, l_{i+1}, \dots, l_j\})(l)(\alpha)) \\
& \quad \text{if } \text{id} \notin \{l_i, l_{i+1}, \dots, l_j\}, \\
& \text{aggregate}(\{l_i, l_{i+1}, \dots, l_j\})(l)(\mu t. \langle l_1 : \tau_1, \dots, l_n : \tau_n \rangle) = \\
& \quad \mu s. \langle l_1 : \tau_1[t \setminus s], \dots, l : \mu t. \langle l_i : \tau_i[t \setminus s], \dots, l_j : \tau_j[t \setminus s] \rangle, \dots, l_n : \tau_n[t \setminus s] \rangle \\
& \quad \text{if } \text{id} \in \{l_i, l_{i+1}, \dots, l_j\} \text{ and } \{l_i, l_{i+1}, \dots, l_j\} \subseteq \{l_1, \dots, l_n\},
\end{aligned}$$

where s is a fresh type variable. For all other cases, we have: $\text{aggregate}(L)(l)(v) = v$. The inverse of aggregation is de-aggregation, defined as a function of type $\mathcal{L} \rightarrow (\text{Types} \rightarrow \text{Types})$:

$$\begin{aligned}
& \text{de_aggregate}(l_i)(\langle l_1 : v_1, \dots, l_n : v_n \rangle) = \\
& \quad \langle l_1 : v_1, \dots, l_{i-1} : v_{i-1}, l'_1 : \sigma_1, \dots, l'_m : \sigma_m, l_{i+1} : v_{i+1}, \dots, l_n : v_n \rangle \\
& \quad \text{if } l_i \in \{l_1, \dots, l_n\} \text{ and } v_i \approx \langle l'_1 : \sigma_1, \dots, l'_m : \sigma_m \rangle \\
& \quad \text{and } \{l'_1, \dots, l'_m\} \cap \{l_1, \dots, l_{i-1}, l_{i+1}, \dots, l_n\} = \emptyset \\
& \text{de_aggregate}(l_i)(\mu t. \langle l_1 : v_1, \dots, l_n : v_n \rangle) = \\
& \quad \mu t. (\text{de_aggregate}(l_i)(\langle l_1 : v_1, \dots, l_n : v_n \rangle)) \\
& \quad \text{if } l_i \in \{l_1, \dots, l_n\} \text{ and } v_i = \langle l'_1 : \sigma_1, \dots, l'_m : \sigma_m \rangle \\
& \quad \text{and } \{l'_1, \dots, l'_m\} \cap \{l_1, \dots, l_{i-1}, l_{i+1}, \dots, l_n\} = \emptyset \\
& \text{de_aggregate}(l_i)(\mu t. \langle l_1 : v_1, \dots, l_n : v_n \rangle) = \\
& \quad \mu t. (\text{de_aggregate}(l_i)(\langle l_1 : v_1, \dots, l_n : v_n \rangle)) \\
& \quad \text{if } l_i \in \{l_1, \dots, l_n\} \text{ and } v_i = \mu s. \langle l'_1 : \sigma_1, \dots, l'_m : \sigma_m \rangle \\
& \quad \text{and } \text{id} \notin \{l'_1, \dots, l'_m\} \text{ and } \{l'_1, \dots, l'_m\} \cap \{l_1, \dots, l_{i-1}, l_{i+1}, \dots, l_n\} = \emptyset \\
& \quad \text{and } s \notin \text{fvars}(\langle l'_1 : \sigma_1, \dots, l'_m : \sigma_m \rangle) \\
& \text{de_aggregate}(l_i)(\mu t. \langle l_1 : v_1, \dots, l_n : v_n \rangle) =
\end{aligned}$$

$$\begin{aligned}
& \mu s. < l_1 : v_1[t \setminus s], \dots, l_{i-1} : v_{i-1}[t \setminus s], \\
& \quad l'_1 : \sigma_1, \dots, l'_m : \sigma_m, l_{i+1} : v_{i+1}[t \setminus s], \dots, l_n : v_n[t \setminus s] > \\
& \text{if } l_i \in \{l_1, \dots, l_n\} \text{ and } v_i = \mu s. < l'_1 : \sigma_1, \dots, l'_m : \sigma_m > \\
& \text{and } \text{id} \in \{l'_1, \dots, l'_m\} \text{ and } \{l'_1, \dots, l'_m\} \cap \{l_1, \dots, l_{i-1}, l_{i+1}, \dots, l_n\} = \emptyset \\
& \text{and } s \notin \text{fvars}(< l'_1 : \sigma_1, \dots, l'_m : \sigma_m >).
\end{aligned}$$

For all other cases, we have: $\text{de_aggregate}(l)(v) = v$. Objectification is defined as a function of type $\text{Types} \rightarrow \text{Types}$:

$$\begin{aligned}
& \text{objectify}(< l_1 : v_1, \dots, l_n : v_n >) = \mu s. < \text{id}:\text{oid}, l_1 : v_1, \dots, l_n : v_n >, \\
& \quad \text{if } \text{id} \notin \{l_1, \dots, l_n\} \\
& \text{objectify}(\mu t. < l_1 : v_1, \dots, l_n : v_n >) = \mu t. < \text{id}:\text{oid}, l_1 : v_1, \dots, l_n : v_n > \\
& \quad \text{if } \text{id} \notin \{l_1, \dots, l_n\},
\end{aligned}$$

where s is a fresh type variable. For all other cases, we have: $\text{objectify}(v) = v$. The inverse of objectify is de_objectify , defined as a function of type $\text{Types} \rightarrow \text{Types}$:

$$\begin{aligned}
& \text{de_objectify}(< \text{id}:\text{oid}, l_1 : v_1, \dots, l_n : v_n >) = < l_1 : v_1, \dots, l_n : v_n > \\
& \text{de_objectify}(\mu t. \alpha) = \mu t. (\text{de_objectify}(\alpha)).
\end{aligned}$$

For all other cases, we have: $\text{de_objectify}(v) = v$. Finally, the set of basic type transformations, denoted by \mathcal{BT} , is given by:

$$\mathcal{BT} = \mathcal{BT}_{\text{ren}} \cup \mathcal{BT}_{\text{agg}} \cup \mathcal{BT}_{\text{obj}},$$

where

$$\begin{aligned}
\mathcal{BT}_{\text{ren}} &= \{\text{rename}(l)(l') \mid l \in \mathcal{L} \wedge l' \in \mathcal{L}\} \\
\mathcal{BT}_{\text{agg}} &= \{\text{tuple}(l) \mid l \in \mathcal{L}\} \cup \{\text{de_tuple}\} \cup \\
& \quad \{\text{aggregate}(L)(l) \mid L \subseteq \mathcal{L} \wedge l \in \mathcal{L}\} \cup \{\text{de_aggregate}(l) \mid l \in \mathcal{L}\} \\
\mathcal{BT}_{\text{obj}} &= \{\text{objectify}, \text{de_objectify}\}.
\end{aligned}$$

□

Note that, ‘renaming’ a field with the name of another field leaves a type unchanged (instead of resulting in an undefined type). In fact, all transformations leave a type unchanged if straight-forward application would result in an undefined type.

Example 9.2 Let σ be type $< \text{name}:\text{string}, \text{age}:\text{integer}, \text{address}:\text{string} >$. Then:

$$\begin{aligned}
& \text{rename}(\text{address})(\text{residence})(\sigma) = \\
& \quad < \text{name}:\text{string}, \text{age}:\text{integer}, \text{residence}:\text{string} > \\
& \text{aggregate}(\{\text{name}, \text{age}\})(\text{person})(\sigma) = \\
& \quad < \text{person}: < \text{name}:\text{string}, \text{age}:\text{integer} >, \text{address}:\text{string} > \\
& \text{objectify}(\sigma) = \\
& \quad \mu s. < \text{id}:\text{oid}, \text{name}:\text{string}, \text{age}:\text{integer}, \text{address}:\text{string} >.
\end{aligned}$$

□

Complex type transformations are obtained by combining basic type transformations.

Definition 9.3 The set of complex type transformations, denoted by \mathcal{CT} , is inductively defined by:

1. if $F \in \mathcal{BT} \cup \{\text{nottransform}\}$, then $F \in \mathcal{CT}$
2. if $F_1 \in \mathcal{CT}$ and $F_2 \in \mathcal{CT}$, then $F_1 \circ F_2 \in \mathcal{CT}$
3. if $F \in \mathcal{CT}$, then $\{F\} \in \mathcal{CT}$
4. if $\{l_1, \dots, l_n\} \subseteq \mathcal{L}$ is a set of n distinct labels and $\{F_1, \dots, F_n\} \subseteq \mathcal{CT}$, then $\langle l_1 : F_1, \dots, l_n : F_n \rangle \in \mathcal{CT}$.

Type transformation *nottransform* leaves all types unchanged:

$$\text{nottransform}(v) = v.$$

Complex type transformation $F_1 \circ F_2$ is the composition of F_1 and F_2 :

$$F_1 \circ F_2(v) = F_1(F_2(v)).$$

Complex type transformation $\{F\}$ transforms set types and leaves other types unchanged:

$$\{F\}(\{v\}) = \{F(v)\}.$$

Complex type transformation $F = \langle l_1 : F_1, \dots, l_n : F_n \rangle$ transforms record types and leaves other types unchanged:

$$\begin{aligned} F(\langle l_1 : v_1, \dots, l_n : v_n \rangle) &= \langle l_1 : F_1(v_1), \dots, l_n : F_n(v_n) \rangle \\ F(\mu t. \langle l_1 : v_1, \dots, l_n : v_n \rangle) &= \mu t. \langle l_1 : F_1(v_1), \dots, l_n : F_n(v_n) \rangle. \end{aligned}$$

In the same way, \mathcal{CT}_{ren} is obtained from \mathcal{BT}_{ren} and \mathcal{CT}_{renagg} from $\mathcal{BT}_{ren} \cup \mathcal{BT}_{agg}$.

□

Example 9.4 Type $\sigma_4 = \langle l_1 : \mu s. \langle \text{id} : \text{oid}, l : \tau_1, l_2 : \tau_2 \rangle, l_3 : \tau_3 \rangle$ can be obtained from type $\sigma_1 = \langle l_1 : \tau_1, l_2 : \tau_2, l_3 : \tau_3 \rangle$ as follows:

$$\begin{aligned} \sigma_2 &= \text{rename}(l_1)(l)(\sigma_1) = \langle l : \tau_1, l_2 : \tau_2, l_3 : \tau_3 \rangle \\ \sigma_3 &= \text{aggregate}(\{l, l_2\})(l_1)(\sigma_2) = \langle l_1 : \langle l : \tau_1, l_2 : \tau_2 \rangle, l_3 : \tau_3 \rangle \\ \sigma_4 &= \langle l_1 : \text{objectify}, l_3 : \text{nottransform} \rangle(\sigma_3). \end{aligned}$$

□

Type transformations preserve derivable equality.

Lemma 9.5 Let τ_1 and τ_2 be types and F be a type transformation in \mathcal{CT}_{renagg} . Then:

$$\tau_1 =_D \tau_2 \Rightarrow F(\tau_1) =_D F(\tau_2).$$

Proof. If $F \in \mathcal{BT}$, then the lemma follows from a simple case distinction w.r.t. F . If $F \in \mathcal{CT} - \mathcal{BT}$, then the lemma follows from an induction argument on the structure of F . \square

Every basic type transformation induces a function between the tree representing the original type and the tree representing the transformed type.

Definition 9.6 Let F be a rename operation $rename(l_1)(l')$ and τ be a type, such that $F(\tau) \neq \tau$. Then F induces a function Φ from the set of paths in $struc(\tau)$ to the set of paths in $struc(F(\tau))$. Function Φ is defined by $\Phi(p) = \varphi(p, \tau)$, where

$$\begin{aligned} \varphi(l, v) &= l' \text{ if } l = l_1 \text{ and } v = \tau \\ \varphi(l, v) &= l \text{ if } l \neq l_1 \text{ or } v \neq \tau \\ \varphi(l.l_1 \dots l_n, \langle \dots, l : \sigma, \dots \rangle) &= \varphi(l, v). \varphi(l_1 \dots l_n, \sigma) \\ \varphi(l.l_1 \dots l_n, \mu t. \langle \dots, l : \sigma, \dots \rangle) &= \\ \varphi(l, v). \varphi(l_1 \dots l_n, \sigma[t \setminus \mu t. \langle \dots, l : \sigma, \dots \rangle]). \end{aligned}$$

Let F be a tupling operation $tuple(l')$ and τ be a type, such that $F(\tau) \neq \tau$. Then F induces a function Φ from the set of paths in $struc(\tau)$ to the set of paths in $struc(F(\tau))$. Function Φ is defined by $\Phi(p) = \varphi(p, \tau)$, where

$$\begin{aligned} \varphi(l, v) &= l'.l \text{ if } v = \tau \\ \varphi(l, v) &= l \text{ if } v \neq \tau \\ \varphi(l.l_1 \dots l_n, \langle \dots, l : \sigma, \dots \rangle) &= \varphi(l, v). \varphi(l_1 \dots l_n, \sigma) \\ \varphi(l.l_1 \dots l_n, \mu t. \langle \dots, l : \sigma, \dots \rangle) &= \\ \varphi(l, v). \varphi(l_1 \dots l_n, \sigma[t \setminus \mu t. \langle \dots, l : \sigma, \dots \rangle]). \end{aligned}$$

Let F be an aggregation operation $aggregate(\{l'_1, \dots, l'_p\})(l')$ and τ be a type, such that $F(\tau) \neq \tau$. Then F induces a function Φ from the set of paths in $struc(\tau)$ to the set of paths in $struc(F(\tau))$. Function Φ is defined by $\Phi(p) = \varphi(p, \tau)$, where

$$\begin{aligned} \varphi(l, v) &= l'.l \text{ if } l \in \{l'_1, \dots, l'_p\} \text{ and } v = \tau \\ \varphi(l, v) &= l \text{ if } l \notin \{l'_1, \dots, l'_p\} \text{ or } v \neq \tau \\ \varphi(l.l_1 \dots l_n, \langle \dots, l : \sigma, \dots \rangle) &= \varphi(l, v). \varphi(l_1 \dots l_n, \sigma) \\ \varphi(l.l_1 \dots l_n, \mu t. \langle \dots, l : \sigma, \dots \rangle) &= \\ \varphi(l, v). \varphi(l_1 \dots l_n, \sigma[t \setminus \mu t. \langle \dots, l : \sigma, \dots \rangle]). \end{aligned}$$

Let F be an objectify operation and τ be a type, such that $F(\tau) \neq \tau$. Then F induces a function Φ from the set of paths in $struc(\tau)$ to the set of paths in $struc(F(\tau))$, which is defined by $\Phi(p) = p$. \square

Every complex type transformation induces a function between the tree representing the original type and the tree representing the transformed type. The function is obtained by combining the functions induced by the basic type transformations that occur in the complex type transformation.

9.2 Transformations on predicates and functions

Type transformations also induce transformations on predicates and functions. The transformation for predicates is obtained by applying the function as defined in Definition 9.6 to the predicate.

Definition 9.7 Let \mathcal{F} be a type transformation, τ be a type, and ψ be a conjunction of formulas, such that each formula has one of the following forms:

1. uniqueness: $\forall x \in e \forall y \in e [(x.p_1 \cong_{TERM} y.p_1 \wedge \dots \wedge x.p_n \cong_{TERM} y.p_n) \Rightarrow x \cong_{TERM} y]$
2. refint: $\forall x \in e \forall x_1 \in x.p_1 \dots \forall x_n \in x_{n-1}.p_n \exists y \in e [x_n.p_{n+1} \cong_{TERM} y]$.

Predicate $\mathcal{F}(\psi)$ is obtained from ψ by replacing each formula in ψ as follows:

1. for each uniqueness formula: for every p_i , if p_i is a path in $struc(\tau)$, then p_i is replaced by $\Phi(p_i)$
2. for each refint formula: for every p_i , if $p_1 \in \dots \in p_i$ is a path in $struc(\tau)$, then p_i is replaced by q_i , such that $\Phi(p_1 \in \dots \in p_i) = \Phi(p_1 \in \dots \in p_{i-1}) \in q_i$,

where Φ is the function from the set of paths in $struc(\tau)$ to the set of paths in $struc(\mathcal{F}(\tau))$ induced by \mathcal{F} . \square

The transformation for function is more involved.

Definition 9.8 Let \mathcal{F} be a type transformation, $\tau = \mu t.\alpha$ and σ be types, and $F = \lambda count:integer \lambda obj:\tau \lambda P.f$ be a function in $functions(\tau, \sigma)$. Function $\mathcal{F}(F)$ is defined as follows:

$$\mathcal{F}(F) = \lambda count:integer \lambda obj:\mathcal{F}(\tau) \lambda P. rep(f, \sigma, \emptyset, \perp, \perp),$$

where

$$\begin{aligned} rep(b, v, V, y, p) &= b \text{ if } b \in Cons \\ rep(x, v, V, y, p) &= x \text{ if } x \in Var \\ rep(l, v, V, y, p) &= l \text{ if } l \in \mathcal{L} \\ rep(obj.l_1 \dots l_n, v, V, y, p) &= obj.\Phi(l_1 \dots l_n) \\ rep(e_1 \theta e_2, v, V, y, p) &= rep(e_1, v, V, y, p) \theta rep(e_2, v, V, y, p) \end{aligned}$$

$$\begin{aligned}
rep(newid(e), v, V, y, p) &= newid(e) \\
rep(e \cup \{e_1, \dots, e_n\}, \{v\}, V, y, p) &= \\
&e \cup \{rep(e_1, v, V, y, p), \dots, rep(e_n, v, V, y, p)\} \\
rep(< l_1 = e_1, \dots, l_n = e_n >, < l_1 : v_1, \dots, l_n : v_n >, V, y, p) &= \\
&< l_1 = rep(e_1, v_1, V, y, p.l_1), \dots, l_n = rep(e_n, v_n, V, y, p.l_n) > \\
rep(\mu x.e, s, V, y, p) &= rep(\mu x.e, \mu s.\beta, V, y, p) \\
&\text{if } \mu s.\beta \in V \\
rep(\mu x.e, \mu s.\beta, V, y, p) &= ap(\mathcal{F}, \mu x.e, \mu s.\beta, V, y, p) \\
&\text{if } s = t \\
rep(\mu x.e, \mu s.\beta, V, y, p) &= \mu x.rep(e, \beta, V \cup \{\mu s.\beta\}, x, empty) \\
&\text{if } s \neq t,
\end{aligned}$$

where Φ is the function from the set of paths in $estruc(\tau)$ to the set of paths in $estruc(\mathcal{F}(\tau))$ induced by \mathcal{F} and $ap(\mathcal{G}, e, v, V, y, p)$ is defined as follows.

Case 1: $\mathcal{G} = rename(l_1)(l)$. Then:

$$\begin{aligned}
ap(\mathcal{G}, < l_1 = e_1, \dots, l_n = e_n >, < l_1 : v_1, \dots, l_n : v_n >, V, y, p) &= \\
&< l = rep(e_1, v_1, V, y, p.l_1), \dots, l_n = rep(e_n, v_n, V, y, p.l_n) > \\
ap(\mathcal{G}, \mu x.e, \mu s.\beta, V, y, p) &= \mu x.ap(\mathcal{G}, e, \beta, V \cup \{\mu s.\beta\}, x, empty).
\end{aligned}$$

For all other cases, we have: $ap(\mathcal{G}, e, v, V, y, p) = rep(e, v, V, y, p)$.

Case 2: $\mathcal{G} = tuple(l)$. Then:

$$ap(\mathcal{G}, \mu x.e, \mu s.\beta, V, y, p) = < l = \mu x.rep(e, \beta, V \cup \{\mu s.\beta\}, x, empty) >.$$

For all other cases, we have: $ap(\mathcal{G}, e, v, V, y, p) = < l = rep(e, v, V, y, p.l) >.$

Case 3: $\mathcal{G} = aggregate(\{l_1, \dots, l_i\})(l)$. Then:

$$\begin{aligned}
ap(\mathcal{G}, < l_1 = e_1, \dots, l_n = e_n >, < l_1 : v_1, \dots, l_n : v_n >, V, y, p) &= \\
&< l = < l_1 = rep(e_1, v_1, V, y, p.l.l_1), \dots, l_i = rep(e_i, v_i, V, y, p.l.l_i) >, \\
&\quad l_{i+1} = rep(e_{i+1}, v_{i+1}, V, y, p.l_{i+1}), \dots, l_n = rep(e_n, v_n, V, y, p.l_n) > \\
ap(\mathcal{G}, \mu x.e, \mu s.\beta, V, y, p) &= \\
&\mu x. ap(\mathcal{G}, e, \beta, V \cup \{\mu s.\beta\}, x, empty) \\
&\text{if } id \notin \{l_1, \dots, l_i\}, \\
ap(\mathcal{G}, \mu x. < l_1 = e_1, \dots, l_n = e_n >, \mu s. < l_1 : v_1, \dots, l_n : v_n >, V, y, p) &= \\
&\mu z_{(y,p)}. < l = \mu x. < l_1 = e'_1, \dots, l_i = e'_i >, l_{i+1} = e''_{i+1}, \dots, l_n = e''_n > \\
&\text{if } id \in \{l_1, \dots, l_i\},
\end{aligned}$$

where $e'_i = rep(e_i[x \setminus z_{(y,p)}], v_i, V \cup \{\mu s. < l_1 : v_1, \dots, l_n : v_n >\}, x, l_i)$ and $e''_i = rep(e_i[x \setminus z_{(y,p)}], v_i, V \cup \{\mu s. < l_1 : v_1, \dots, l_n : v_n >\}, z_{(y,p)}, l_i)$ and $z_{(y,p)}$ is an instance variable that does not occur in F and uniquely depends on both y and p . For all other cases, we have: $ap(\mathcal{G}, e, v, V, y, p) = rep(e, v, V, y, p)$.

Case 4: $\mathcal{G} = objectify$. Then:

$$ap(\mathcal{G}, < l_1 = e_1, \dots, l_n = e_n >, < l_1 : v_1, \dots, l_n : v_n >, V, y, p) =$$

$$\begin{aligned}
& \mu z_{(y,p)}. < \text{id} = \text{newid}(\text{count} + i_{(y,p)}), \\
& \quad l_1 = \text{rep}(e_1, v_1, V, z_{(y,p)}, l_1), \dots, l_n = \text{rep}(e_n, v_n, V, z_{(y,p)}, l_n) > \\
& \text{if } \text{id} \notin \{l_1, \dots, l_n\} \\
& \text{ap}(\mathcal{G}, \mu x. < l_1 = e_1, \dots, l_n = e_n >, \mu s. < l_1 : v_1, \dots, l_n : v_n >, V, y, p) = \\
& \quad \mu x. < \text{id} = \text{newid}(\text{count} + i_{(y,p)}), \\
& \quad \quad l_1 = \text{rep}(e_1, v_1, V', x, l_1), \dots, l_n = \text{rep}(e_n, v_n, V', x, l_n) > \\
& \text{if } \text{id} \notin \{l_1, \dots, l_n\},
\end{aligned}$$

where $V' = V \cup \{\mu s. < l_1 : v_1, \dots, l_n : v_n >\}$ and $z_{(y,p)}$ is an instance variable that does not occur in F and uniquely depends on both y and p , and $i_{(y,p)}$ is an integer that uniquely depends on both y and p , such that $\text{newid}(\text{count} + i_{(y,p)})$ does not occur in F . For all other cases, we have: $\text{ap}(\mathcal{G}, e, v, V, y, p) = \text{rep}(e, v, V, y, p)$.

Case 5: $\mathcal{G} = \{\mathcal{G}'\}$. Then:

$$\begin{aligned}
& \text{ap}(\mathcal{G}, e \cup \{e_1, \dots, e_n\}, \{v\}, V, y, p) = \\
& \quad e \cup \{\text{ap}(\mathcal{G}', e_1, v, V, y, p), \dots, \text{ap}(\mathcal{G}', e_n, v, V, y, p)\}.
\end{aligned}$$

For all other cases, we have: $\text{ap}(\mathcal{G}, e, v, V, y, p) = \text{rep}(e, v, V, y, p)$.

Case 6: $\mathcal{G} = < l_1 : \mathcal{G}_1, \dots, l_n : \mathcal{G}_n >$. Then:

$$\begin{aligned}
& \text{ap}(\mathcal{G}, < l_1 = e_1, \dots, l_n = e_n >, < l_1 : v_1, \dots, l_n : v_n >, V, y, p) = \\
& \quad < l_1 = \text{ap}(\mathcal{G}_1, e_1, v, V, y, p), \dots, l_n = \text{ap}(\mathcal{G}_n, e_n, v, V, y, p) > \\
& \text{ap}(\mathcal{G}, \mu x. e, \mu s. \beta, V, y, p) = \\
& \quad \mu x. \text{ap}(\mathcal{G}, e, \beta, V \cup \{\mu s. \beta\}, x, \text{empty}).
\end{aligned}$$

For all other cases, we have: $\text{ap}(\mathcal{G}, e, v, V, y, p) = \text{rep}(e, v, V, y, p)$. \square

Unfortunately, not every transformed function is well-typed. Therefore, we introduce well-typed functions.

Definition 9.9 Let \mathcal{F} be a type transformation, $\tau = \mu t. \alpha$ and σ be types, and $F = \lambda \text{count} : \text{integer} \lambda \text{obj} : \tau \lambda P. f$ be a function in $\text{functions}(\tau, \sigma)$. Furthermore let $F' = \lambda \text{count} : \text{integer} \lambda \text{obj} : \mathcal{F}(\tau) \lambda P. f'$ be $\mathcal{F}(F)$. Then:

$$\text{well-typed}(F') \Leftrightarrow \text{typed}(f', \bar{\mathcal{F}}(\sigma), \emptyset),$$

where $\bar{\mathcal{F}}(\sigma)$ is obtained by applying \mathcal{F} to the right components of σ :

$$\begin{aligned}
& \bar{\mathcal{F}}(B) = B \text{ if } B \in BTypes \\
& \bar{\mathcal{F}}(s) = s \text{ if } s \in TypeVar \\
& \bar{\mathcal{F}}(\{v\}) = \{\bar{\mathcal{F}}(v)\} \\
& \bar{\mathcal{F}}(< l_1 : v_1, \dots, l_n : v_n >) = < l_1 : \bar{\mathcal{F}}(v_1), \dots, l_n : \bar{\mathcal{F}}(v_n) > \\
& \bar{\mathcal{F}}(\mu s. \beta) = \mathcal{F}(\mu s. \beta) \\
& \quad \text{if } s = t \\
& \bar{\mathcal{F}}(\mu s. \beta) = \mu s. \mathcal{F}(\beta) \\
& \quad \text{if } s \neq t
\end{aligned}$$

and $typed(e, v, V)$, which is true if and only if e has type v , is inductively defined as follows:

$typed(b, B, V)$ if $b \in Cons_B$
 $typed(x, s, V)$ if $x \in Var_s$
 $typed(l, B, V)$ if $l:B$ is a parameter in P
 $typed(obj.l_1 \dots l_n, v, V)$
 if $estruc(\mathcal{F}(\tau).l_1 \dots l_n) = estruc(v, V)$
 $typed(e_1 \theta e_2, v, V)$
 if $typed(e_1, v, V)$ and $typed(e_2, v, V)$ and $\theta \in ops_B$
 $typed(newid(e), v, V)$
 if $v = oid$
 $typed(e \cup \{e_1, \dots, e_n\}, \{v\}, V)$
 if $typed(e, \{v\}, V)$ and for every $i \in \{1, \dots, n\}$, $typed(e_i, v, V)$
 $typed(< l_1=e_1, \dots, l_n=e_n >, < l_1:v_1, \dots, l_n:v_n >, V)$
 if for every $i \in \{1, \dots, n\}$, $typed(e_i, v_i, V)$
 $typed(\mu x.e, s, V)$
 if $\mu s.\beta \in V$ and $typed(\mu x.e, \mu s.\beta, V)$
 $typed(\mu x.e, \mu s.\beta, V)$
 if $x \in Var_s$ and $typed(e, \beta, V \cup \{\mu s.\beta\})$.

□

Finally, we can define transformations on classes.

Definition 9.10 Let H be a well-defined class hierarchy and C be a class in H . Furthermore let $class$ be $(type(C), constraint(C, e), funcs(C))$ and \mathcal{F} be a type transformation. Then \mathcal{F} induces the following transformation on classes: follows:

$$\mathcal{F}(class) = (\mathcal{F}(type(C)), \mathcal{F}(constraint(C, e)), \{\mathcal{F}(F) \mid F \in funcs(C)\}).$$

Class $\mathcal{F}(class)$ is well-defined if $\mathcal{F}(F)$ is well-typed for every $F \in funcs(C)$. □

Example 9.11 The following class hierarchy introduces a class Person, a class Employee, which inherits from class Person, and a class Company:

```

Class Person
Attributes
  name : string
  street : string
  house : integer
  city : string
Endclass
Class Employee Isa Person
Attributes

```

```

    employer : Company
    salary : integer
Endclass
Class Company
Attributes
    name : string
Endclass.

```

The underlying type of class Person is:

```

 $\mu$  Person. <id:oid, name:string, street:string, house:integer, city:string>.

```

The underlying type of class Person can be transformed into (using $F_1 = \text{aggregate}(\{\text{street}, \text{house}, \text{city}\}) (\text{address})$):

```

 $\mu$  Person. <id:oid, name:string, address:
    <street:string, house:integer, city:string>>,

```

which can be transformed into (using $F_2 = \langle \text{id:} \text{nottransform}, \text{name:} \text{nottransform}, \text{address:} \text{objectify } \rangle$):

```

 $\mu$  Person. <id:oid, name:string, address:
     $\mu$  Address. <id: oid, street:string, house:integer, city:string>>.

```

The composite transformation ($F = F_2 \circ F_1$) is a variant of the transformation for lexical attributes from [28]. We can redefine class Person as a class (named Person1) that refers to a new class (named Address).

```

Class Person1
Attributes
    name : string
    address : Address
Endclass
Class Address
Attributes
    street : string
    house : integer
    city : string
Endclass.

```

Since the identities of the objects in class Person become the identities of the objects in the redefined class (Person1), the underlying constraint of class Person (*constr*) is preserved by the redefined class (w.r.t. F): the underlying constraint of the redefined class implies $F(\text{constr})$. The underlying type of class Employee is:

μ Employee. <id:oid, name:string, street:string, house:integer, city:string,
employer: τ_C , salary:integer>,

where τ_C is the underlying type of class Company. The underlying type of class Employee can be transformed into (using $F_1 = \text{aggregate}(\{\text{id}, \text{name}, \text{street}, \text{house}, \text{city}\})$ (employee)):

μ Works_for. <employee: μ Employee. <id:oid, name:string,
street:string, house:integer, city:string>,
employer: τ_C , salary:integer>,

which can be transformed into (using $F_2 = \text{objectify}$):

μ Works_for. <id:oid, employee: μ Employee. <id:oid, name:string,
street:string, house:integer, city:string>,
employer: τ_C , salary:integer>.

The composite transformation ($F = F_2 \circ F_1$) is a variant of the transformation for unstable subtypes from [28]. We can redefine class Employee as a ‘relation’ (named Works_for) that refers to a new class (named Employee1):

```

Class Works_for
Attributes
  employee : Employee1
  employer : Company
  salary   : integer
Endclass
Class Employee1
Attributes
  name : string
  street : string
  house : integer
  city : string
Endclass.

```

Since the identities of the objects in class Employee become the identities of the objects in class Employee1, and not the identities of the objects in the redefined class (Works_for), the underlying constraint of class Employee (*constr*) is not preserved by the redefined class (w.r.t. F): the underlying constraint of the redefined class and the rule ‘ $x.\text{employee.id} = y.\text{employee.id} \Rightarrow x.\text{employee} = y.\text{employee}$ ’ do not imply $F(\text{constr})$. Therefore, we introduce a key for class Works_for:

```

Class Works_for1
Attributes
  employee : Employee1

```

```
    employer : Company
    salary : integer
Constraints
    key employee
Endclass.
```

The underlying constraint of this class and the rule ' $x.\text{employee.id} = y.\text{employee.id}$
 $\Rightarrow x.\text{employee} = y.\text{employee}$ ' do imply $F(\text{constr})$. \square

Chapter 10

Properties of schema transformations

In this Chapter, we consider the transformations of the previous chapter. We introduce transformational type equivalence based on renaming and aggregation operations, and semantic type equivalence based on data capacity. Furthermore, we prove that the set of renaming and aggregation operations is sound and complete w.r.t. data capacity by proving that transformational and semantic type equivalence are equivalent. This result is important for the comparison of attributes, because we know that underlying types with the same data capacity can be obtained by a combination of renaming and aggregation operations.

10.1 Normal forms

We only consider underlying types of classes and their component types, because, if both $\mu t.\alpha$ and $\mu t.\beta$ occur in such a type, then we know that $\alpha = \beta$, which simplifies the definitions of transformational and semantic type equivalence.

First, we define transformational type equality and transformational type equivalence in terms of renaming and aggregation operations.

Definition 10.1 Let τ_1 and τ_2 be closed types. Transformational equality of τ_1 and τ_2 , denoted by $\tau_1 =_{trans} \tau_2$, is defined as follows:

$$\tau_1 =_{trans} \tau_2 \Leftrightarrow \exists F \in CT_{renagg} [F(\tau_1) =_D \tau_2].$$

Transformational equivalence, denoted by \cong_{trans} , is defined by:

$$\tau_1 \cong_{trans} \tau_2 \Leftrightarrow \exists v_1 \in Types \exists v_2 \in Types [\tau_1 \cong_D v_1 \wedge \tau_2 \cong_D v_2 \wedge v_1 =_{trans} v_2].$$

□

Transformational equality is a symmetric relation.

Lemma 10.2 Let τ_1 and τ_2 be types. Then:

$$\exists F_1 \in \mathcal{CT} [F_1(\tau_1) =_D \tau_2] \Rightarrow \exists F_2 \in \mathcal{CT} [F_2(\tau_2) =_D \tau_1].$$

Proof. Suppose $F_1 \in \mathcal{CT}$ and $F_1(\tau_1) =_D \tau_2$. If $F_1 \in \mathcal{BT}$, then it follows from a simple case distinction w.r.t. F_1 that there is a transformation $F_2 \in \mathcal{CT}$ such that $F_2(\tau_2) =_D \tau_1$. And if $F_1 \in \mathcal{CT} - \mathcal{BT}$, then it follows from an induction argument on the structure of F_1 that there is a transformation $F_2 \in \mathcal{CT}$ such that $F_2(\tau_2) =_D \tau_1$. \square

Example 10.3 Let σ_0 be type $\mu t. \langle \text{id:oid, name:string, spouse:t} \rangle$. Then:

$$\begin{aligned} \sigma_1 &= \text{aggregate}(\{\text{id, name}\})(\text{person})(\sigma_0) \\ &= \mu s. \langle \text{person:}\mu t. \langle \text{id:oid, name:string} \rangle, \text{spouse:s} \rangle \\ \sigma_2 &= \text{de_aggregate}(\text{person})(\sigma_1) \\ &= \sigma_0. \end{aligned}$$

\square

Using Lemma 10.2, we can deduce that for every combination of a type transformation and a type there exists an inverse type transformation.

Lemma 10.4 Let F be a type transformation and τ be a type. Then there exists a type transformation F' , such that $F'(F(\tau)) = \tau$.

Proof. Let τ' be $F(\tau)$. According to Lemma 10.2, there exists a type transformation F' , such that $F'(\tau') =_D \tau$. Hence, $F'(F(\tau)) = F'(\tau') =_D \tau$. \square

Transformational type equivalence is weaker than the notions of type equivalence of Chapter 4.

Lemma 10.5 Let τ_1 and τ_2 be types. Then:

$$\tau_1 \cong_D \tau_2 \Rightarrow \tau_1 \cong_{\text{trans}} \tau_2.$$

Proof.

$$\begin{aligned} \tau_1 \cong_D \tau_2 &\Rightarrow \\ \tau_1 \cong_R \tau_2 &\Rightarrow \\ \text{rd}(\tau_1) =_D \text{rd}(\tau_2) &\Rightarrow \\ \text{notransform}(\text{rd}(\tau_1)) =_D \text{rd}(\tau_2) &\Rightarrow \\ \text{rd}(\tau_1) =_{\text{trans}} \text{rd}(\tau_2) &\Rightarrow \\ \tau_1 \cong_D \text{rd}(\tau_1) =_{\text{trans}} \text{rd}(\tau_2) \cong_D \tau_2 &\Rightarrow \\ \tau_1 \cong_{\text{trans}} \tau_2. & \end{aligned}$$

\square

Types can be rewritten by applying de-tupling and de-aggregation operations.

Definition 10.6 Let τ be a type. The flattened form of τ , denoted by $nf(\tau)$, is defined as follows:

$$\begin{aligned}
 nf(t) &= t & \text{if } t \in \text{TypeVar} \\
 nf(B) &= B & \text{if } B \in \text{BTypes} \\
 nf(\{v\}) &= \{nf(v)\} \\
 nf(< l_1 : v_1, \dots, l_n : v_n >) &= \\
 &\quad \text{collapse}(< l_1 : nf(v_1), \dots, l_n : nf(v_n) >) \\
 nf(\mu t. \alpha) &= \mu t. (nf(\alpha)) & \text{if } t \in \text{fvars}(\alpha) \\
 nf(\mu t. \alpha) &= nf(\alpha) & \text{if } t \notin \text{fvars}(\alpha),
 \end{aligned}$$

where $\text{collapse}(\tau')$ is obtained from τ' by applying the following rewrite rules until they cannot be applied any more:

$$\begin{aligned}
 1. & < l_1 : v_1, \dots, l_i : < l'_1 : \sigma_1, \dots, l'_k : \sigma_k >, \dots, l_n : v_n > \longrightarrow \\
 & < l_1 : v_1, \dots, \nu(\{l_1, \dots, l_n\} - \{l_i\}, l'_1) : \sigma_1, \dots, \\
 & \quad \nu(\{l_1, \dots, l_n\} - \{l_i\}, l'_k) : \sigma_k, \dots, l_n : v_n > \\
 2. & < l : v > \longrightarrow v
 \end{aligned}$$

where ν is a function from $\wp(\mathcal{L}) \times \mathcal{L}$ to \mathcal{L} , such that $\nu(L, l) = l$ if $l \notin L$ and $\nu(L, l) \notin L$ if $l \in L$. \square

Since every type has exactly one flattened form, the rewrite process is a normalisation process and the resulting flattened forms are normal forms (different from the normal forms of Chapter 4). Furthermore, the normalisation process preserves derivable equality.

Lemma 10.7 Let τ_1 and τ_2 be types. Then:

$$\tau_1 =_D \tau_2 \Rightarrow nf(\tau_1) =_D nf(\tau_2).$$

Proof. If $F \in \mathcal{BT}$, then the lemma follows from a simple case distinction w.r.t. F . If $F \in \mathcal{CT} - \mathcal{BT}$, the lemma follows from an induction argument on the structure of F . \square

The following lemma gives the relationship between transformational equality and normal forms.

Lemma 10.8 Let τ_1 and τ_2 be closed types. Then:

$$\tau_1 =_{\text{trans}} \tau_2 \Leftrightarrow nf(\tau_1) \sim nf(\tau_2),$$

where \sim is defined as follows: $v_1 \sim v_2 \Leftrightarrow \exists F \in \mathcal{CT}_{\text{ren}}[F(v_1) =_D v_2]$.

Proof of \Rightarrow . Let τ be a type and F be a type transformation in $\mathcal{BT}_{\text{renagg}} = \mathcal{BT}_{\text{ren}} \cup \mathcal{BT}_{\text{agg}}$. By a simple case distinction w.r.t. F , it follows that:

$$nf(\tau) \sim nf(F(\tau)).$$

Let τ be a type and F be a type transformation in $\mathcal{CT}_{renagg} - \mathcal{BT}_{renagg}$. By a simple induction on the structure of F , it follows that:

$$nf(\tau) \sim nf(F(\tau)).$$

Since $\tau_1 =_{trans} \tau_2$, there is a type transformation F , such that $F(\tau_1) =_D \tau_2$. Hence:

$$nf(\tau_1) \sim nf(F(\tau_1)) =_D nf(\tau_2).$$

From the definition of \sim , it follows that: $nf(\tau_1) \sim nf(\tau_2)$.

Proof of \Leftarrow . Since $nf(\tau_1) \sim nf(\tau_2)$, there is a renaming operation $F \in \mathcal{CT}_{ren}$, such that $F(nf(\tau_1)) =_D nf(\tau_2)$. Note that every application of a rewrite rule in the normalisation process can be obtained by a combination of a number of renaming operations and a de-aggregation operation. Hence, there are type transformations F_1 and F_2 , such that:

$$F(F_1(\tau_1)) =_D F_2(\tau_2).$$

Furthermore, from Lemma 10.4 we can deduce that there is a type transformation F'_2 , such that $F'_2(F_2(\tau_2)) =_D \tau_2$. It follows that:

$$F'_2(F(F_1(\tau_1))) =_D F'_2(F_2(\tau_2)) =_D \tau_2.$$

Hence, $\tau_1 =_{trans} \tau_2$. \square

10.2 Soundness

In this section, we define semantic type equivalence and prove soundness of transformational type equivalence. To define semantic type equivalence, we have to define applied type variables, preterms, data capacity functions, and unfolded forms. The set of applied type variables of a type is given by the following definition.

Definition 10.9 Let τ be a type. The set of applied type variables of τ , denoted by $avars(\tau)$, is defined as:

$$\begin{aligned} avars(t) &= \{t\} \text{ if } t \in TypeVar \\ avars(B) &= \emptyset \text{ if } B \in BTypes \\ avars(\{v\}) &= avars(v) \\ avars(< l_1 : v_1, \dots, l_m : v_m >) &= avars(v_1) \cup \dots \cup avars(v_m) \\ avars(\mu t. \alpha) &= avars(\alpha). \end{aligned}$$

\square

The set of preterms of a type is the set of terms of depth 1.

Definition 10.10 First, for every basic type B and every natural number n , we postulate $\text{Cons}(B, n)$ as a subset of Cons_B consisting of n elements, and, for every type variable t and every natural number n , we postulate $\text{Var}(t, n)$ as a subset of Var_t consisting of n elements.

Let τ be a type, such that $\text{avars}(\tau) = \{t_{i_1}, \dots, t_{i_n}\}$, where $i_1 < \dots < i_n$. Furthermore, let $\vec{p} = (p_1, p_2, p_3, p_4)$ and $\vec{q} = (q_1, \dots, q_n)$ be natural number vectors. The set of preterms of τ w.r.t. \vec{p} and \vec{q} , denoted by $\text{preterms}(\tau, \vec{p}, \vec{q})$, is defined as follows:

$$\begin{aligned} \text{preterms}(t_{i_j}, \vec{p}, \vec{q}) &= \text{Var}(t_{i_j}, q_j) \text{ if } t_{i_j} \in \text{TypeVar}, \\ \text{preterms}(B_i, \vec{p}, \vec{q}) &= \text{Cons}(B_i, p_i) \text{ if } B_i \in \text{BTypes}, \\ \text{preterms}(\{v\}, \vec{p}, \vec{q}) &= \{ \{e_1, \dots, e_n\} \mid \forall i \in \{1, \dots, n\} [e_i \in \text{preterms}(v, \vec{p}, \vec{q})] \}, \\ \text{preterms}(< l_1 : v_1, \dots, l_n : v_n >, \vec{p}, \vec{q}) &= \\ &\{ < l_1 = e_1, \dots, l_n = e_n > \mid \forall i \in \{1, \dots, n\} [e_i \in \text{preterms}(v_i, \vec{p}, \vec{q})] \}, \\ \text{preterms}(\mu t. \alpha, \vec{p}, \vec{q}) &= \{ \mu x. e \mid x \in \text{Var}(t, 1) \wedge e \in \text{preterms}(\alpha, \vec{p}, \vec{q}) \}, \end{aligned}$$

where B_1 denotes type oid, B_2 denotes type integer, B_3 denotes type rational, and B_4 denotes type string. \square

Note that every set of preterms is finite. The data capacity function of a type is defined as follows.

Definition 10.11 Let τ be a type, such that $\text{avars}(\tau) = \{t_{i_1}, \dots, t_{i_n}\}$, where $i_1 < \dots < i_n$. The data capacity function of type τ , denoted by χ_τ , is defined as:

$$\lambda \vec{p} \lambda \vec{q}. \chi(\tau, \vec{p}, \vec{q}),$$

where $\vec{p} = (p_1, p_2, p_3, p_4)$ and $\vec{q} = (q_1, \dots, q_n)$ are natural number vectors and:

$$\begin{aligned} \chi(t_{i_j}, \vec{p}, \vec{q}) &= q_j \text{ if } t_{i_j} \in \text{TypeVar}, \\ \chi(B_i, \vec{p}, \vec{q}) &= p_i \text{ if } B_i \in \text{BTypes}, \\ \chi(\{v\}, \vec{p}, \vec{q}) &= 2^{\chi(v, \vec{p}, \vec{q})}, \\ \chi(< l_1 : v_1, \dots, l_m : v_m >, \vec{p}, \vec{q}) &= \chi(v_1, \vec{p}, \vec{q}) \times \dots \times \chi(v_m, \vec{p}, \vec{q}), \\ \chi(\mu t. \alpha, \vec{p}, \vec{q}) &= \chi(\alpha, \vec{p}, \vec{q}), \end{aligned}$$

where B_1 denotes type oid, B_2 denotes type integer, B_3 denotes type rational, and B_4 denotes type string. \square

Lemma 10.12 Let τ be a type. Then data capacity function χ_τ is injective in every parameter.

Proof. From a simple induction argument on the structure of τ , it follows that $\lambda \vec{p} \lambda \vec{q}. \chi(\tau, \vec{p}, \vec{q})$ is strictly increasing in every parameter p_i and every parameter q_i . \square

The data capacity function of a type gives the number of preterms of the type.

Lemma 10.13 Let τ be a type, such that $avars(\tau) = \{t_{i_1}, \dots, t_{i_n}\}$, where $i_1 < \dots < i_n$. For every natural number vector $\vec{p} = (p_1, p_2, p_3, p_4)$ and every natural number vector $\vec{q} = (q_1, \dots, q_n)$:

$$\chi_\tau(\vec{p}, \vec{q}) = |preterms(\tau, \vec{p}, \vec{q})|.$$

Proof. Using the definition of preterms and the fact that every set of preterms is finite, we can deduce that:

$$|preterms(\{v\}, \vec{p}, \vec{q})| = |\wp_{fin}(preterms(v, \vec{p}, \vec{q}))| = |\wp(preterms(v, \vec{p}, \vec{q}))|.$$

The lemma follows from this fact and an induction argument on the structure of τ (cf. [1]). \square

The unfolded form of a type w.r.t. a type variable t is obtained by replacing every occurrence of t by the type that defines t (i.e., starts with μt).

Definition 10.14 Let τ be a type. Finally, the unfolded form of τ w.r.t. t , denoted by $unfold(\tau, t)$, is defined as:

$$\begin{aligned} unfold(\{v\}, t) &= \{unfold(v, t)\} \\ unfold(<\dots, l : v, \dots>, t) &= <\dots, unfold(v, t), \dots> \\ unfold(\mu t.\alpha, t) &= elim(\alpha[t \setminus \mu t.\alpha], \emptyset) \\ unfold(\mu t'.\alpha', t) &= \mu t'.(unfold(\alpha', t)) \text{ if } t' \neq t, \end{aligned}$$

where $t \in bvars(v)$ and $t \in bvars(\alpha')$, and $elim$ eliminates the second occurrence of $\mu t'$ whenever one $\mu t'$ occurs in the range of another $\mu t'$:

$$\begin{aligned} elim(t', V) &= t' && \text{if } t' \in TypeVar \\ elim(B, V) &= B && \text{if } B \in BTypes \\ elim(\{v\}, V) &= \{elim(v, V)\} \\ elim(<l_1 : v_1, \dots, l_m : v_m>, V) &= \\ &<l_1 : elim(v_1, V), \dots, l_m : elim(v_m, V)> \\ elim(\mu t'.\alpha', V) &= elim(\alpha', V) \\ &\text{if } t' \in V \\ elim(\mu t'.\alpha', V) &= \mu t'.(elim(\alpha, V \cup \{t'\})) \\ &\text{if } t' \notin V. \end{aligned}$$

\square

Semantic type equality and semantic type equivalence are defined in terms of data capacity functions.

Definition 10.15 Let τ_1 and τ_2 be types. Semantic equality of τ_1 and τ_2 , denoted by $\tau_1 =_{sem} \tau_2$, is defined as follows:

$$\tau_1 =_{sem} \tau_2 \Leftrightarrow \exists v_1 [\tau_1 =_D v_1 \wedge \chi_{v_1} = \chi_{\tau_2} \wedge \forall t \in avars(v_1) [\chi_{unfold(v_1, t)} = \chi_{unfold(\tau_2, t)}]].$$

Semantic equivalence, denoted by \cong_{sem} , is defined by:

$$\tau_1 \cong_{sem} \tau_2 \Leftrightarrow \exists v_1 \in Types \exists v_2 \in Types [\tau_1 \cong_D v_1 \wedge \tau_2 \cong_D v_2 \wedge v_1 =_{sem} v_2],$$

where $=$ denotes extensional equality of functions, which is the standard notion of equality for functions. \square

Semantic type equivalence is weaker than the notions of type equivalence in Chapter 4 (cf. Lemma 10.5).

Example 10.16 Let σ_1 be $\mu t_1. < a: < a_1:integer, a_2:\{t_1\} >, b:string > \text{ and } \sigma_2$ be $\mu t_2. < a:integer, b:\{t_2\}, c:string >$. Then $\sigma_1 \cong_{sem} \sigma_2$, because $\sigma_1 =_{sem} \sigma_2$:

$$\begin{aligned} \chi_{\sigma_1} &= \chi_{\sigma_2} = \lambda \vec{p} \lambda q. (p_2 \times 2^q \times p_4) \\ \chi_{unfold(\sigma_1, t_1)} &= \chi_{unfold(\sigma_2, t_2)} = \lambda \vec{p} \lambda q. (p_2 \times 2^{(p_2 \times 2^q \times p_4)} \times p_4). \end{aligned}$$

However, $\sigma_1 \not\cong_D \sigma_2$. \square

Finally, we prove that transformational equality is sound w.r.t. semantic equality.

Theorem 10.17 Let τ_1 and τ_2 be closed types. Then:

$$\tau_1 =_{trans} \tau_2 \Rightarrow \tau_1 =_{sem} \tau_2.$$

Proof. Let τ be a type and F be a basic type transformation. By a case distinction w.r.t. F , it follows that:

$$F(\tau) =_{sem} \tau.$$

Let τ be a type and F be a complex type transformation in \mathcal{CT}_{renagg} . By an induction on the structure of F , it follows that:

$$F(\tau) =_{sem} \tau.$$

Now, suppose $\tau_1 =_{trans} \tau_2$, i.e., there is a type transformation $F \in \mathcal{CT}_{renagg}$, such that $F(\tau_1) =_D \tau_2$. Then:

$$\tau_1 =_{sem} F(\tau_1) =_D \tau_2.$$

From the definition of $=_{sem}$, it follows that: $\tau_1 =_{sem} \tau_2$. \square

From this theorem and the definitions of transformational and semantic equivalence, it follows that transformational equivalence is sound w.r.t. semantical equivalence.

10.3 Completeness

In this section, we prove that transformational type equivalence is complete with respect to semantic type equivalence.

The proof of completeness is based on a number of lemmas which will be proven first. To prove the lemmas, we have to define the head of a type and the tails of a type.

Definition 10.18 Let τ be a type. The head of τ , denoted by $hd(\tau)$, is defined as:

$$\begin{aligned} hd(t) &= t \text{ if } t \in TypeVar \\ hd(B) &= B \text{ if } B \in BTypes \\ hd(\{v\}) &= \{hd(v)\} \\ hd(< l_1 : v_1, \dots, l_m : v_m >) &= < l_1 : hd(v_1), \dots, l_m : hd(v_m) > \\ hd(\mu t. \alpha) &= t. \end{aligned}$$

Let t be a bound type variable in τ . The tail of τ w.r.t. t , denoted by $tl(\tau, t)$, is defined as:

$$\begin{aligned} tl(\{v\}, t) &= tl(v, t) \\ tl(< \dots, l : v, \dots >, t) &= tl(v, t) \\ tl(\mu t. \alpha, t) &= \mu t. \alpha \\ tl(\mu t'. \alpha', t) &= tl(\alpha', t) \text{ if } t' \neq t \end{aligned}$$

where $t \in bvars(v)$ and $t \in bvars(\alpha')$. \square

The following lemma gives the relationship between the head and the tails of a type.

Lemma 10.19 Let σ be a type and V be $avars(hd(\sigma)) \cap bvars(\sigma)$. Then:

$$hd(\sigma)[t \setminus tl(\sigma, t) \mid t \in V] = \sigma.$$

Proof.

The lemma follows from an induction argument on the structure of type σ . If σ is a basic type or a type variable, then $V = \emptyset$ and, hence, $hd(\sigma)[t \setminus tl(\sigma, t) \mid t \in V] = \sigma$.

If $\sigma = \{\sigma'\}$, then:

$$\begin{aligned} hd(\sigma)[t \setminus tl(\sigma, t) \mid t \in V] &= \\ \{hd(\sigma')\}[t \setminus tl(\sigma', t) \mid t \in V] &= \\ \{hd(\sigma')[t \setminus tl(\sigma', t) \mid t \in V]\} &= \\ \{\sigma'\} &= \sigma. \end{aligned}$$

If $\sigma = < l_1 : \sigma_1, \dots, l_n : \sigma_n >$, then:

$$\begin{aligned}
& hd(\sigma)[t \setminus tl(\sigma, t) \mid t \in V] = \\
& < l_1 : hd(\sigma_1), \dots, l_n : hd(\sigma_n) > [t \setminus tl(\sigma, t) \mid t \in V] = \\
& < l_1 : hd(\sigma_1)[t \setminus tl(\sigma_1, t) \mid t \in V_1], \dots, l_n : hd(\sigma_n)[t \setminus tl(\sigma_n, t) \mid t \in V_n] > = \\
& < l_1 : \sigma_1, \dots, l_n : \sigma_n > = \sigma.
\end{aligned}$$

If $\sigma = \mu t'. \alpha$, then $V = \{t'\}$ and $hd(\sigma)[t' \setminus tl(\sigma, t')] = t'[t' \setminus \sigma] = \sigma$. \square

The following two lemmas give the relationship between the data capacity functions of a type and the unfolded form of the type.

Lemma 10.20 Let σ be a type and t be a type variable in $bvars(\sigma)$, such that $t = t_j$ for some j . Then:

$$\lambda \vec{p} \lambda \vec{q}. \chi_{unfold(\sigma, t)}(\vec{p}, \vec{q}) = \lambda \vec{p} \lambda \vec{q}. \chi_{\sigma}(\vec{p}, \vec{q}[q_j \setminus \chi_{tl(\sigma, t)}(\vec{p}, \vec{y})]),$$

where \vec{y} contains the type variables that occur in $tl(\sigma, t)$.

Proof.

The lemma follows from an induction argument on the structure of type σ . We give a proof for the non-trivial case: $\sigma = \mu t. \alpha$. Let there be n occurrences of t in α . Let α' be α , with every occurrence of t replaced by a unique member of $\{t_{i_1}, \dots, t_{i_n}\}$, which is a set of new type variables. Then:

$$\begin{aligned}
\chi_{unfold(\sigma, t)} &= \chi_{elim(\alpha[t \setminus \sigma])} = \\
&\chi_{\alpha'[t_{i_1} \setminus elim(\sigma, V_1), \dots, t_{i_n} \setminus elim(\sigma, V_n)]} = \\
&\lambda \vec{p} \lambda \vec{q}. (\chi_{\alpha'}(\vec{p}, \vec{y})[q_{i_1} \setminus \chi_{\sigma}(\vec{p}, \vec{q}), \dots, q_{i_n} \setminus \chi_{\sigma}(\vec{p}, \vec{q})]) = \\
&\lambda \vec{p} \lambda \vec{q}. (\chi_{\alpha}(\vec{p}, \vec{q})[q_j \setminus \chi_{\sigma}(\vec{p}, \vec{q})]) = \lambda \vec{p} \lambda \vec{q}. (\chi_{\sigma}(\vec{p}, \vec{q})[q_j \setminus \chi_{\sigma}(\vec{p}, \vec{q})]),
\end{aligned}$$

where the V_i 's are the sets induced by applying $elim$, and \vec{y} contains the type variables that occur in α' . \square

Lemma 10.21 Let σ and σ' be types and t be a type variable in $bvars(\sigma) \cap bvars(\sigma')$. Then:

$$(\chi_{\sigma} = \chi_{\sigma'} \wedge \chi_{unfold(\sigma, t)} = \chi_{unfold(\sigma', t)}) \Rightarrow \chi_{tl(\sigma, t)} = \chi_{tl(\sigma', t)}.$$

Proof.

Suppose $\chi_{\sigma} = \chi_{\sigma'}$ and $\chi_{unfold(\sigma, t)} = \chi_{unfold(\sigma', t)}$. Furthermore, suppose $\chi_{tl(\sigma, t)} \neq \chi_{tl(\sigma', t)}$. Then:

$$\begin{aligned}
\lambda \vec{p} \lambda \vec{q}. \chi_{unfold(\sigma, t)}(\vec{p}, \vec{q}) &= \\
&\lambda \vec{p} \lambda \vec{q}. \chi_{\sigma}(\vec{p}, \vec{q}[q_j \setminus \chi_{tl(\sigma, t)}(\vec{p}, \vec{y})]) \neq \\
&\lambda \vec{p} \lambda \vec{q}. \chi_{\sigma}(\vec{p}, \vec{q}[q_j \setminus \chi_{tl(\sigma', t)}(\vec{p}, \vec{y})]) = \\
&\lambda \vec{p} \lambda \vec{q}. \chi_{\sigma'}(\vec{p}, \vec{q}[q_j \setminus \chi_{tl(\sigma', t)}(\vec{p}, \vec{y})]) = \lambda \vec{p} \lambda \vec{q}. \chi_{unfold(\sigma', t)}(\vec{p}, \vec{q}),
\end{aligned}$$

where the first step follows from Lemma 10.20, the second from the fact that χ_{σ} is injective in q_j and the fact that $\chi_{tl(\sigma, t)} \neq \chi_{tl(\sigma', t)}$, the third from the fact that $\chi_{\sigma} = \chi_{\sigma'}$, and the fourth from Lemma 10.20. Contradiction. Hence, $\chi_{tl(\sigma, t)} = \chi_{tl(\sigma', t)}$. \square

The following three lemmas give the relationship between the data capacity functions of a number of types on one hand and the data capacity function of the combined type obtained by substitution on the other hand.

Lemma 10.22 Let σ be a type and t be a type variable in $fvars(\sigma)$, such that $t = t_j$ for some j . Furthermore, let τ be a type. Then:

$$\chi_{\sigma[t \setminus \tau]} = \lambda \vec{p} \lambda \vec{z}. (\chi_{\sigma}(\vec{p}, \vec{q})[q_j \setminus \chi_{\tau}(\vec{p}, \vec{y})]),$$

where \vec{y} contains the type variables that occur in τ and \vec{z} contains the type variables that occur in $\sigma[t \setminus \tau]$.

Proof.

The lemma follows from an induction argument on the structure of type σ . We give a proof for the non-trivial case: $\sigma = t$. Then $\chi_{\sigma[t \setminus \tau]} = \chi_{\tau}$ and:

$$\chi_{\sigma}(\vec{p}, \vec{q})[q_j \setminus \chi_{\tau}(\vec{p}, \vec{y})] = q_j[q_j \setminus \chi_{\tau}(\vec{p}, \vec{y})] = \chi_{\tau}(\vec{p}, \vec{y}).$$

Since $\vec{z} = \vec{y}$ in this case, it follows that:

$$\lambda \vec{p} \lambda \vec{z}. (\chi_{\sigma}(\vec{p}, \vec{q})[q_j \setminus \chi_{\tau}(\vec{p}, \vec{y})]) = \lambda \vec{p} \lambda \vec{z}. \chi_{\tau}(\vec{p}, \vec{z}) = \chi_{\tau}.$$

□

Lemma 10.23 Let σ and σ' be types and t be a type variable in $fvars(\sigma) \cap fvars(\sigma')$, such that $t = t_j$ for some j . Furthermore, let τ , and τ' be types, such that $t \in avars(\tau) \cap avars(\tau')$. Then:

$$(\chi_{\sigma[t \setminus \tau]} = \chi_{\sigma'[t \setminus \tau']} \wedge \chi_{\tau} = \chi_{\tau'}) \Rightarrow \chi_{\sigma} = \chi_{\sigma'}.$$

Proof.

Suppose $\chi_{\sigma[t \setminus \tau]} = \chi_{\sigma'[t \setminus \tau']}$ and $\chi_{\tau} = \chi_{\tau'}$. Furthermore, suppose $\chi_{\sigma} \neq \chi_{\sigma'}$. Then $\lambda q_j. \chi_{\sigma}(\vec{p}, \vec{q}) \neq \lambda q_j. \chi_{\sigma'}(\vec{p}, \vec{q})$. Without loss of generality, let Q be a natural number, such that:

$$\forall q_j \geq Q [\chi_{\sigma}(\vec{p}, \vec{q}) > \chi_{\sigma'}(\vec{p}, \vec{q})].$$

Since $\forall q_j [\chi_{\tau}(\vec{p}, \vec{q}) > q_j]$, it follows that:

$$a) \forall q_j \geq Q [\chi_{\sigma}(\vec{p}, \vec{q}[q_j \setminus \chi_{\tau}(\vec{p}, \vec{y})]) > \chi_{\sigma'}(\vec{p}, \vec{q}[q_j \setminus \chi_{\tau}(\vec{p}, \vec{y})])].$$

Then:

$$\begin{aligned} \chi_{\sigma[t \setminus \tau]} &= \lambda \vec{p} \lambda \vec{z}. (\chi_{\sigma}(\vec{p}, \vec{q}[q_j \setminus \chi_{\tau}(\vec{p}, \vec{y})])) \neq \\ &\quad \lambda \vec{p} \lambda \vec{z}. (\chi_{\sigma'}(\vec{p}, \vec{q}[q_j \setminus \chi_{\tau}(\vec{p}, \vec{y})])) = \\ &\quad \lambda \vec{p} \lambda \vec{z}. (\chi_{\sigma'}(\vec{p}, \vec{q}[q_j \setminus \chi_{\tau'}(\vec{p}, \vec{y})])) = \chi_{\sigma'[t \setminus \tau]}, \end{aligned}$$

where the first step follows from Lemma 10.22, the second from a), the third from the fact that $\chi_{\tau} = \chi_{\tau'}$, and the fourth from Lemma 10.22. Contradiction. Hence, $\chi_{\sigma} = \chi_{\sigma'}$. □

Lemma 10.24 Let σ and σ' be types, such that $fvars(\sigma) = fvars(\sigma') = \{t_i \mid i \in I\}$, where I is a subset of the natural numbers. Furthermore, let $\{\sigma_i \mid i \in I\}$ and $\{\sigma'_i \mid i \in I\}$ be sets of types, such that $t_i \in avars(\sigma_i) \cap avars(\sigma'_i)$. Define:

$$\begin{aligned}\tau &= \sigma[i \setminus \sigma_i \mid i \in I] \\ \tau' &= \sigma'[i \setminus \sigma'_i \mid i \in I].\end{aligned}$$

Then:

$$(\chi_\tau = \chi_{\tau'} \wedge \forall i \in I [\chi_{\sigma_i} = \chi_{\sigma'_i}]) \Rightarrow \chi_\sigma = \chi_{\sigma'}.$$

Proof.

The lemma follows from $|I|$ applications of Lemma 10.23. \square

The following lemma states that the data capacity functions corresponding to the different type variables in a type uniquely determine the levels at which the type variables are bound.

Lemma 10.25 Let σ and σ' be types, such that $bvars(\sigma) = bvars(\sigma') = \{t_i \mid i \in I\}$, where I is a subset of the natural numbers. Furthermore, let V be $avars(hd(\sigma)) \cap bvars(\sigma)$ and V' be $avars(hd(\sigma')) \cap bvars(\sigma')$. For every $i \in I$, define:

$$\begin{aligned}\mu t_i. \alpha_i &= tl(\sigma, t_i) \\ \mu t_i. \alpha'_i &= tl(\sigma', t_i).\end{aligned}$$

Then:

$$\forall i \in I [\chi_{\alpha_i} = \chi_{\alpha'_i}] \Rightarrow V = V'.$$

Proof.

Suppose $\forall i \in I [\chi_{\alpha_i} = \chi_{\alpha'_i}]$. In order to prove the lemma, it is sufficient to prove that α_i contains $\mu t_j. \alpha_j$ if and only if α'_i contains $\mu t_j. \alpha'_j$, where type σ_1 contains type σ_2 if σ_2 occurs in σ_1 .

Suppose α_i contains $\mu t_j. \alpha_j$. It follows that χ_{α_i} contains χ_{α_j} , where function χ_1 contains function χ_2 if χ_1 is the composition of χ_2 and another function χ : $\chi_1 = \chi \circ \chi_2$. Hence, $\chi_{\alpha'_i} = \chi_{\alpha_i}$ has at least one occurrence of χ_j . Suppose α'_i does not contain $\mu t_j. \alpha'_j$. Since $\chi_{\alpha'_i}$ has at least one occurrence of χ_j , α'_j must contain $\mu t_i. \alpha_i$. Then $\chi_{\alpha_j} = \chi_{\alpha'_j}$ contains $\chi_{\alpha'_i}$. It follows that $\chi_{\alpha_i} \neq \chi_{\alpha'_i}$. Contradiction. Hence, α'_i contains $\mu t_j. \alpha'_j$. \square

Finally, we can prove that transformational equality is complete w.r.t. semantic equality.

Theorem 10.26 Let τ_1 and τ_2 be closed types. Then:

$$\tau_1 =_{sem} \tau_2 \Rightarrow \tau_1 =_{trans} \tau_2.$$

Proof.

For non-recursive types, the theorem follows from Theorem 5.4 in [1]. For recursive types, the proof is more involved.

Suppose $\tau_1 =_{sem} \tau_2$. Let v_1 be the type, such that $v_1 =_D \tau_1$ and $\chi_{v_1} = \chi_{\tau_2}$. For every i in $I = \{j \mid t_j \in avars(v_1)\}$, define:

$$\begin{aligned} \mu t_i. \alpha_i &= tl(v_1, t_i) \\ \mu t_i. \alpha'_i &= tl(\tau_2, t_i) \\ V_i &= avars(hd(\alpha_i)) \cap bvars(\alpha_i). \end{aligned}$$

From $\chi_{v_1} = \chi_{\tau_2}$, $\forall i \in I [\chi_{unfold(v_1, t_i)} = \chi_{unfold(\tau_2, t_i)}]$, Lemma 10.21 and 10.25, it follows that, for every i in I :

$$\begin{aligned} \text{b) } \chi_{\alpha_i} &= \chi_{\alpha'_i} \\ \text{c) } V_i &= avars(hd(\alpha'_i)) \cap bvars(\alpha'_i). \end{aligned}$$

For every i in I , define $\sigma_i = \mu t_i. \alpha_i$ and $\sigma'_i = \mu t_i. \alpha'_i$. Using c) and Lemma 10.19, we obtain:

$$\begin{aligned} \alpha_i &= hd(\alpha_i)[t_j \setminus tl(\alpha_i, t_j) \mid t_j \in V_i] \\ \alpha'_i &= hd(\alpha'_i)[t_j \setminus tl(\alpha'_i, t_j) \mid t_j \in V_i]. \end{aligned}$$

From this, b), and Lemma 10.24, it follows that:

$$\chi_{hd(\alpha_i)} = \chi_{hd(\alpha'_i)}.$$

Using the fact that $hd(\alpha_i)$ and $hd(\alpha'_i)$ are non-recursive types, we can conclude that:

$$\text{d) } nf(hd(\alpha_i)) \sim nf(hd(\alpha'_i)).$$

By an induction argument on the number of nestings of μ -operators, we can prove that, for every $i \in I$, $nf(\sigma_i) \sim nf(\sigma'_i)$. The induction step of the induction argument is:

$$\begin{aligned} nf(\alpha_i) &= nf(hd(\alpha_i)[t_j \setminus tl(\alpha_i, t_j) \mid t_j \in V_i]) = \\ &= nf(hd(\alpha_i)[t_j \setminus \sigma_j \mid t_j \in V_i]) = \\ &= (nf(hd(\alpha_i)))[t_j \setminus nf(\sigma_j) \mid t_j \in V_i] \sim \\ &= (nf(hd(\alpha'_i)))[t_j \setminus nf(\sigma'_j) \mid t_j \in V_i] = \\ &= nf(hd(\alpha'_i)[t_j \setminus \sigma'_j \mid t_j \in V_i]) = \\ &= nf(hd(\alpha'_i)[t_j \setminus tl(\alpha'_i, t_j) \mid t_j \in V_i]) = \\ &= nf(\alpha'_i), \end{aligned}$$

where the first step follows from Lemma 10.19; the second from the definition of tl and the fact that v_1 contains α_i ; the third from the definition of nf and the fact that every t_j occurs free in α_j ; the fourth from the induction hypothesis and d); the fifth from the definition of nf and the fact that every t_j occurs free in α_j ; the

sixth from the definition of tl and the fact that τ_2 contains α'_i ; and the final from Lemma 10.19 and c).

From $v_1 = hd(v_1)[t_j \setminus \mu t_j. \alpha_j \mid t_j \in V]$, where $V = avars(hd(v_1)) \cap bvars(v_1)$, Lemma 10.19 and 10.25, it follows that:

$$\tau_2 = hd(\tau_2)[t_j \setminus \mu t_j. \alpha'_j \mid t_j \in V].$$

Using $\chi_{v_1} = \chi_{\tau_2}$, b), and Lemma 10.24, we can conclude that $\chi_{hd(v_1)} = \chi_{hd(\tau_2)}$. Hence, $nf(hd(v_1)) \sim nf(hd(\tau_2))$ and, because of the fact that $nf(\sigma_j) \sim nf(\sigma'_j)$, $nf(v_1) \sim nf(\tau_2)$. That is, $\tau_1 =_{trans} \tau_2$. \square

From this theorem and the definitions of transformational and semantic equivalence, it follows that transformational equivalence is complete w.r.t. semantical equivalence. Using the result of the previous section, we can conclude that transformational equivalence is sound and complete w.r.t. semantical equivalence.

Chapter 11

Combined approach

In this chapter, we adapt the approach of Chapters 6 and 7 to cope with keys, query methods, method calls, and object creation. Again, we want to compare and integrate schemas by their real world semantics, where attributes, constraints, and query methods correspond to different kinds of structural (i.e., state-related) properties and update methods correspond to behavioural (i.e., transition-related) properties. It follows that we can compare and integrate classes by integrating attributes, constraints, update methods, and query methods separately. For that purpose, we adapt the subclass relation of Chapter 6 and the join operator of Chapter 7.

Furthermore, we extend the approach using schema transformations to detect schemas that are the same modulo a transformation and we show that syntactic properties of methods can be used to choose among a set of integration mappings or a set of schema transformations.

11.1 Comparison of classes

To adapt the subclass relation, we have to consider comparison of attributes, comparison of keys, and comparison of methods. The comparison of attributes is exactly the same as in Chapter 6, based on the subtype morphisms of that Chapter. The comparison of keys is just testing for equality. The motivation for this choice is that keys cannot imply other keys (cf. Armstrong's axioms [62]).

To adapt the comparison of methods, we have to redefine specialisations of functional forms and generalisations of methods. The specialisation of a functional form of a method is obtained by extending it to the type of a subclass.

Definition 11.1 Let H be a well-defined class hierarchy, C_1 and C_2 be classes in H , and φ be a subtype morphism from $type(C_1)$ to $type(C_2)$. Furthermore, let

$meth$ be an update method in $u_meths(C_1)$ and $F = \lambda count:integer \lambda obj:type(C_1) \lambda P.f$ be $func_u(C_1, meth)$. Functional form $\mathcal{S}_\varphi^u(F)$ is defined by:

$$\lambda obj:type(C_2) \lambda P. \mathcal{S}_\varphi(f, empty),$$

where \mathcal{S}_φ is the same as in Definition 6.11 of Chapter 6, with the following additional item:

$$\mathcal{S}_\varphi(newid(e), p) = newid(e)$$

Now, let $meth = m(P \rightarrow l:T) = E$ be a query method in $q_meths(C_1)$ and $F = \lambda count:integer \lambda obj:type(C_1) \lambda P.f$ be $func_q(C_1, meth)$. Functional form $\mathcal{S}_\varphi^q(F)$ is defined by:

$$\lambda obj:type(C_2) \lambda P. \mathcal{S}_{\hat{\varphi}(T)}(f, empty),$$

where

$$\begin{aligned} \hat{\varphi}(B) &= id_{estruc}(B) \text{ if } B \in \{\text{integer, rational, string}\} \\ \hat{\varphi}(\{U\}) &= \{\hat{\varphi}(U)\} \\ \hat{\varphi}(< l_1:U_1, \dots, l_n:U_n >) &= < l_1 = \hat{\varphi}(U_1), \dots, l_n = \hat{\varphi}(U_n) > \\ \hat{\varphi}(name(C_1)) &= \varphi \\ \hat{\varphi}(name(D)) &= id_{estruc}(type(D)) \\ &\text{if } D \in H \text{ and } name(D) \neq name(C_1). \end{aligned}$$

□

The generalisation of a method is obtained by projecting it on a superclass.

Definition 11.2 Let H be a well-defined class hierarchy, C_1 and C_2 be classes in H , and φ be a subtype morphism from $type(C_1)$ to $type(C_2)$. The generalisation of $m(P) = E$ in C_1 , denoted by $\varphi^{-1}(m(P) = E)$, is defined as:

$$m(\tilde{P}) = \varphi^{-1}(E),$$

and the generalisation of $m(P \rightarrow l:T) = E$ in C_1 , denoted by $\varphi^{-1}(m(P \rightarrow l:T) = E)$, is defined as:

$$m(\tilde{P} \rightarrow l : \varphi^{-1}(T)) = \varphi^{-1}(E),$$

where φ^{-1} is the same as in Chapter 6, with the following additional items:

$$\begin{aligned} \varphi^{-1}(B) &= B \text{ if } B \in \{\text{integer, rational, string}\} \\ \varphi^{-1}(\{U\}) &= \{\varphi^{-1}(U)\} \\ \varphi^{-1}(< l_1:U_1, \dots, l_n:U_n >) &= < l_1:\varphi^{-1}(U_1), \dots, l_n:\varphi^{-1}(U_n) > \\ \varphi^{-1}(name(C_2)) &= name(C_1) \\ \varphi^{-1}(name(D)) &= name(D) \\ &\text{if } D \in H \text{ and } name(D) \neq name(C_2) \end{aligned}$$

$$\begin{aligned}
\varphi^{-1}(l.l_1 \cdots l_n) &= l.S_{\varphi^{-1}(T)}(l_1 \cdots l_n) \\
&\text{if } l \text{ is the name of the result} \\
\varphi^{-1}(p.m(p_1, \dots, p_n)) &= \varphi^{-1}(p).m(\varphi^{-1}(p_1), \dots, \varphi^{-1}(p_n)) \\
\varphi^{-1}(\mathbf{new}(\text{name}(C_2), a_1 = p_1, \dots, a_n = p_n)) &= \\
&\quad \mathbf{new}(\text{name}(C_1), \varphi^{-1}(a_i) = \varphi^{-1}(p_i) \mid i \in \{1, \dots, n\} \wedge \exists T[a_i:T \in \text{atts}(C_1)]) \\
\varphi^{-1}(\mathbf{new}(\text{name}(D), a_1 = p_1, \dots, a_n = p_n)) &= \\
&\quad \mathbf{new}(\text{name}(C_2), a_1 = p_1, \dots, a_n = p_n) \\
&\quad \text{if } D \in H \wedge \text{name}(D) \neq \text{name}(C_2) \\
\varphi^{-1}(\mathbf{nsself}) &= \mathbf{nsself}
\end{aligned}$$

and \tilde{P} is obtained from P by removing parameters that do not occur in $\varphi^{-1}(E)$.
 \square

Now we can redefine subclass morphisms.

Definition 11.3 Let H be a well-defined class hierarchy, C_1 and C_2 be classes in H , φ be a type morphism from $\text{type}(C_1)$ to $\text{type}(C_2)$, ψ_u be a function from $u_funcs(C_1)$ to $u_funcs(C_2)$, and ψ_q be a function from $q_funcs(C_1)$ to $q_funcs(C_2)$, where

$$\begin{aligned}
u_funcs(C) &= \{func_u(C, meth) \mid meth \in u_meths(C)\} \\
q_funcs(C) &= \{func_q(C, meth) \mid meth \in q_meths(C)\}.
\end{aligned}$$

The triple $(\varphi, \psi_u, \psi_q)$ is a subclass morphism from C_1 to C_2 if and only if:

1. φ is a subtype morphism
2. $\forall p \in keys(C_1)[\Phi(p) \in keys(C_2)]$
3. $\forall f \in u_funcs(C_1)[\psi_u(f) \leq_{FUNC}^w S_\varphi^u(f)]$
4. $\forall f \in q_funcs(C_1)[\psi_q(f) \leq_{FUNC}^w S_\varphi^q(f)]$

where Φ is the function from the set of paths in $struc(\text{type}(C_1))$ to the set of paths in $struc(\text{type}(C_2))$ induced by φ . Finally, C_1 is a weak subclass of C_2 , denoted by $C_1 \preceq_{CLASS}^w C_2$, if and only if there is a subclass morphism from C_2 to C_1 . \square

A method is specialised if the functional form of its generalisation is the same as the generalisation of its functional form.

Definition 11.4 Let H be a well-defined class hierarchy, C_1 and C_2 be classes in H , and φ be a subtype morphism from $\text{type}(C_1)$ to $\text{type}(C_2)$. Furthermore, let $meth$ be an update method in C_2 . Then $meth$ is a specialised method according to φ , denoted by $spec_u(meth, \varphi)$, if and only if:

$$func_u(C_2, meth) \leq_{FUNC}^w S_\varphi^u(func_u(C_1, \varphi^{-1}(meth))).$$

Now, let $meth$ be a query method in C_2 . Then $meth$ is a specialised method according to φ , denoted by $spec_q(meth, \varphi)$, if and only if:

$$func_q(C_2, meth) \leq_{FUNC}^w S_\varphi^q(func_q(C_1, \varphi^{-1}(meth))).$$

□

11.2 Integration of classes

To adapt the join operator, we have to consider joins of attributes, joins of keys, and joins of methods. The attribute join is exactly the same as in Chapter 7, based on the integration mappings and the type joins of that Chapter. The key join is the intersection of the set of generalised keys.

Definition 11.5 Let $H = \{C_1, \dots, C_n\}$ be a well-defined class hierarchy and Φ be an integration mapping for H as defined in Chapter 7. Furthermore, let K_i be $keys(C_i)$ and K_j be $keys(C_j)$. The join of K_i and K_j according to Φ , denoted by $K_i \sqcup_\Phi K_j$, is given by:

$$K_i \sqcup_\Phi K_j = \{\psi_{i,j}^{-1}(k) \mid k \in K_i \wedge \psi_{i,j}^{-1}(k) \neq \perp\} \cap \{\psi_{j,i}^{-1}(k) \mid k \in K_j \wedge \psi_{j,i}^{-1}(k) \neq \perp\}.$$

□

To adapt the method join, we have to redefine the generalisation of a method set.

Definition 11.6 Let H be a class hierarchy as in Definition 11.5, C_i and C_j be classes in H , U_i be $u_meths(C_i)$, and U_j be $u_meths(C_j)$. Furthermore, let Φ be an integration mapping for H , $\tilde{C}_{(i,j)}$ be $(\nu(\{c_i, c_j\}), \emptyset, \sqcup_\Phi(\{A_i, A_j\}), \emptyset, \emptyset)$, $\psi_{i,j}$ be the subtype morphism from $type(\tilde{C}_{(i,j)})$ to $type(C_i)$ induced by $\varphi_{i,j}$ and α (as in Chapter 7), and $\psi_{j,i}$ be the subtype morphism from $type(\tilde{C}_{(i,j)})$ to $type(C_j)$ induced by $\varphi_{j,i}$ and α (as in Chapter 7). The generalisation of U_i in $\tilde{C}_{(i,j)}$, denoted by $\psi_{i,j}^{-1}(U_i)$ and the generalisation of U_j in $\tilde{C}_{(i,j)}$, denoted by $\psi_{j,i}^{-1}(U_j)$, are defined by:

$$\begin{aligned} \psi_{i,j}^{-1}(U_i) &= \{\psi_{i,j}^{-1}(meth) \mid meth \in U_i \wedge spec_u(meth, \psi_{i,j})\} \\ \psi_{j,i}^{-1}(U_j) &= \{\psi_{j,i}^{-1}(meth) \mid meth \in U_j \wedge spec_u(meth, \psi_{j,i})\}. \end{aligned}$$

The generalisations of the sets of query methods of class C_i and class C_j are defined in exactly the same way. □

The join of a pair of method sets according to an integration mapping is obtained by combining the generalisations of the method sets.

Definition 11.7 Let H be a class hierarchy as in Definition 11.5, $CN(H)$ be the set of class names in H , $MN(H)$ be the set of method names in H , and $\mu : \wp(CN(H) \times MN(H)) \rightarrow CN(H) \times MN(H)$ be an injective choice function for joins of methods, such that $\mu(V) \in V$ for every $V \in CN(H) \times MN(H)$. Furthermore, let C_i and C_j be classes in H , U_i be $u_meths(C_i)$, U_j be $u_meths(C_j)$, $\psi_{i,j}$ and $\psi_{j,i}$ be as defined in the previous definition, and

$$\begin{aligned} C_{(i,j)} &= (\nu(\{c_i, c_j\}), \emptyset, \sqcup_{\Phi}(\{A_i, A_j\}), \emptyset, \psi_{i,j}^{-1}(U_i)) \\ C_{(j,i)} &= (\nu(\{c_i, c_j\}), \emptyset, \sqcup_{\Phi}(\{A_i, A_j\}), \emptyset, \psi_{j,i}^{-1}(U_j)). \end{aligned}$$

The join of U_i and U_j according to Φ , denoted by $U_i \sqcup_{\Phi} U_j$, is given by:

$$\begin{aligned} U_i \sqcup_{\Phi} U_j &= \{meth \mid m_i(P_i) = E_i \in \psi_{i,j}^{-1}(U_i) \wedge m_j(P_j) = E_j \in \psi_{j,i}^{-1}(U_j) \wedge \\ &\quad func_u(C_{(i,j)}, m_i(P_i) = E_i) \stackrel{w}{=}_{FUNC} func_u(C_{(j,i)}, m_j(P_j) = E_j) \wedge \\ &\quad (\mu(\{(c_i, m_i), (c_j, m_j)\}) = (c_i, m_i) \Rightarrow meth = m_i(P_i) = E_i) \wedge \\ &\quad (\mu(\{(c_i, m_i), (c_j, m_j)\}) = (c_j, m_j) \Rightarrow meth = m_j(P_j) = E_j)\}. \end{aligned}$$

The join of the sets of query methods of class C_i and class C_j is defined in exactly the same way. \square

The pre-join of a pair of classes according to an integration mapping is defined in terms of the attribute join, the key join, and the method join.

Definition 11.8 Let $H = \{C_1, \dots, C_n\}$ be a well-defined class hierarchy, where $c_i = name(C_i)$, $A_i = atts(C_i)$, $K_i = keys(C_i)$, $U_i = u_meths(C_i)$, and $Q_i = q_meths(C_i)$. Furthermore, let Φ be an integration mapping. The join of C_i and C_j according to Φ , denoted by $C_i \sqcup_{\Phi} C_j$ is given by:

$$(\nu(\{c_i, c_j\}), \emptyset, A_i \sqcup_{\Phi} A_j, K_i \sqcup_{\Phi} K_j, (U_i \sqcup_{\Phi} U_j) \cup (Q_i \sqcup_{\Phi} Q_j)).$$

\square

Finally, join classes, pre-join hierarchies, and join hierarchies are exactly the same as in Chapter 7.

11.3 Application of schema transformations

Now, we define the set of factors of a class. A factor of a class C is a class that contains a part of the attributes, keys, and methods of C .

Definition 11.9 Let \mathcal{H} be the set of well-defined class hierarchies defined in Chapter 8. Furthermore, let H be a hierarchy in \mathcal{H} and C be a class in H . The set of factors of class C is defined as:

$$\begin{aligned} factors(C) &= \{(d, \emptyset, A, K, M) \mid d \in CN \wedge A \subseteq atts(C) \wedge K \subseteq keys(C) \wedge \\ &\quad M \subseteq meths(C) \wedge H \cup \{(d, \emptyset, A, K, M)\} \in \mathcal{H}\}. \end{aligned}$$

□

Second, we give an example to illustrate that a class can be transformed in several ways, using different factors and different transformations.

Example 11.10 Let class Employee be the following class:

```

Class Employee
Attributes
  name : string
  dob : Date
  street : string
  house : integer
  city : string
  employer : Company
Methods
  move (s:string,h:integer,c:string) =
    street := s; house := h; city := c
Endclass

```

and class Address be a factor of Employee:

```

Class Address
Attributes
  street : string
  house : integer
  city : string
Methods
  move (s:string,h:integer,c:string) =
    street := s; house := h; city := c
Endclass.

```

One option to transform class Employee is to redefine Employee as a subclass of Address (factorisation by specialisation):

```

Class Employee1 Isa Address
Attributes
  name : string
  dob : Date
  employer: Company
Endclass.

```

Another option is to redefine Employee as a class referring to Address (factorisation by delegation):

```

Class Employee2
Attributes
  name : string
  dob : Date
  address: Address2
  employer: Company
Methods
  move (s:string,h:integer,c:string) =
    address := address.new_address(s,h,c)
Endclass
Class Address2
Attributes
  street : string
  house : integer
  city : string
Methods
  new_address (s:string,h:integer,c:string → l:Address2) =
    l := new(Address2, street=s, house=h, city=c)
Endclass.

```

Note that, as an employee is not an address in the real world, it is unlikely that the first option is the right choice. The second option, where employee refers to an address (as one of its attributes) is a more reasonable choice. Now, let class Person be a factor of class Employee2:

```

Class Person
Attributes
  name : string
  dob : Date
  address : Address2
Methods
  move (s:string,h:integer,c:string) =
    address := address.new_address(s,h,c)
Endclass.

```

One option to transform class Employee2 is to redefine Employee2 as a subclass of Person (factorisation by specialisation):

```

Class Employee3 Isa Person
Attributes
  employer : Company
Endclass.

```

Another option is to redefine Employee2 as a class referring to Person (factorisation by delegation):

```

Class Employee4
Attributes
  person : Person1
  employer : Company
Methods
  move (s:string,h:integer,c:string) =
    person := person.new_person(s,h,c)
Endclass
Class Person1
Attributes
  name : string
  dob : Date
  address : Address2
Methods
  new_person (s:string,h:integer,c:string → l:Person1) =
    l := new(Person1, name=name, dob=dob,
      address=address.new_address(s,h,c))
Endclass.

```

Since the identities of the objects in class Employee2 become the identities of the objects in class Employee4, we redefine method ‘move’ to be applicable to objects in class Employee4.

Yet another option is to redefine class Employee2 as a relation involving class Person:

```

Class Employment
Attributes
  employee : Person
  employer : Company
Constraints
  key employee
Endclass.

```

Since the identities of the objects in class Employee2 become the identities of the objects in class Person, we do not redefine method ‘move’, because it is already applicable to objects in class Person.

Note that, as an employee is a person in the real world, it is likely that options one and three are more reasonable than option two, where an employer refers to a person (as one of its attributes). □

As we have seen, a class can be transformed in several ways, using different factors and different transformations, e.g., factorisation by specialisation, factorisation by delegation, or redefinition as a relation. But how do we choose factors and how do we choose between specialisation, delegation and redefinition as a relation?

For that purpose, we introduce evidence ratios for relatedness. Weak relatedness for a set of attributes says whether the attributes are mutually related (according to the methods). Strong relatedness for a set of attributes says whether the attributes are mutually related, but not to attributes outside the set (according to the methods). Isolation for a set of attributes says whether the attributes are not related to attributes outside the set (according to the methods).

Definition 11.11 Let H be a well-defined class hierarchy, C be a class in H , c be $\text{name}(C)$, and M be $\text{meths}(C)$. Furthermore, for $\text{meth} \in M$, let $\text{atts}(\text{meth})$ consist of the names of attributes of C that occur in meth . Weak relatedness of a set of attributes $A \subseteq \{a \mid a : T \in \text{atts}(C)\}$ is defined as:

$$\text{weakrel}(c, A) = \frac{|\{\text{meth} \in M \mid \text{atts}(\text{meth}) \supseteq A\}|}{|\{\text{meth} \in M \mid \text{atts}(\text{meth}) \cap A \neq \emptyset\}|}.$$

Strong relatedness of a set of attributes A is defined as:

$$\text{strongrel}(c, A) = \frac{|\{\text{meth} \in M \mid \text{atts}(\text{meth}) = A\}|}{|\{\text{meth} \in M \mid \text{atts}(\text{meth}) \cap A \neq \emptyset\}|}.$$

Isolation of a set of attributes $A \subseteq \{a \mid a : T \in \text{atts}(C)\}$ is defined as:

$$\text{isolation}(c, A) = \frac{|\{\text{meth} \in M \mid \emptyset \neq \text{atts}(\text{meth}) \subseteq A\}|}{|\{\text{meth} \in M \mid \text{atts}(\text{meth}) \cap A \neq \emptyset\}|}.$$

If $\{\text{meth} \in M \mid \text{atts}(\text{meth}) \cap A \neq \emptyset\}$ is empty, then $\text{weakrel}(c, A)$ and $\text{strongrel}(c, A)$ are defined to be 0, and $\text{isolation}(c, A)$ is defined to be 1.

For a set of attributes with strong relatedness ratio 1 and any method, either all attributes occur in the method and all attributes that occur in the method are in the set, or no attribute in the set occurs in the method. In that case, the attributes are strongly related. For a set of attributes with weak relatedness ratio 0, there is no method in which all attributes occur and, hence, the attributes are not (mutually) related. And for a set of attributes with isolation ratio 1 and any method, either all attributes that occur in the method are attributes in the set or no attribute that occurs in the method is an attribute in the set. In that case, the attributes are only related within the set. \square

Weak and strong relatedness can help to choose a factor. If the strong relatedness ratio of a set of attributes is high, then it is reasonable to believe that they belong together and, hence, to factorise. On the other hand, if the weak relatedness ratio is low, then it is reasonable to believe that they do not belong together and, hence, not to factorise.

Example 11.12 Consider class Employee of Example 11.10. The weak and strong relatedness ratios for street, house, city and name, dob are given by:

$strongrel(Employee, \{street, house, city\}) = 1$
 $weakrel(Employee, \{street, house, city\}) = 1$
 $strongrel(Employee, \{name, dob\}) = 0$
 $weakrel(Employee, \{name, dob\}) = 0.$

As we can see, street, house, and city are strongly related, whereas name and dob are not related.

Now, consider class Employee2 of Example 11.10. The weak and strong relatedness ratios for name, dob, address and name, dob, employer are given by:

$strongrel(Employee2, \{name, dob, address\}) = 0$
 $weakrel(Employee2, \{name, dob, address\}) = 0$
 $strongrel(Employee2, \{name, dob, employer\}) = 0$
 $weakrel(Employee2, \{name, dob, employer\}) = 0.$

As we can see, in both cases the attributes are not related. \square

Isolation can help to choose between specialisation and redefinition as a relation. If the isolation ratio is less than one, then specialisation is possible, but redefinition as a relation is not, since, in that case, we have to add a method to the relation that updates another relation or class.

Example 11.13 Consider class Employee2 of Example 11.10. The isolation evidence ratio for name, dob, address is given by:

$isolation(Employee2, \{name, dob, address\}) = 1.$

Redefinition as a relation results in a relation (Employment) that represents a simple association between a person and a company. Now, if we add a method to class Employee2 that updates attribute address and attribute employer, then we will have to add a method to Employment that creates a new person and updates attribute employee and attribute employer. However, this method inserts objects into a class different from the relation and should therefore not be associated with the relation. \square

So, how do we choose factors and transformations? Factors are chosen by comparing weak evidence ratios. If the weak evidence ratio of a set of attributes is greater than some threshold, there is reason to assume that the attributes can be used as a factor. If not, there is no reason. Transformations are chosen by comparing strong evidence ratios and isolation ratios. In case the strong evidence ratio is greater than some threshold, delegation is a reasonable option, because the attributes are strongly related within the set and weakly related with other attributes. In case the isolation ratio is less than one, then specialisation is possible, but redefinition as a relation is not. Otherwise, specialisation or redefinition as a relation are both possible. It should be mentioned that, in the context of

schema integration, schema transformations must be applied carefully and only if necessary. In particular, this is true for factorisation by specialisation, since a lot of new classes will be generated by this type of transformation.

The considerations for choosing factors and transformations can be used in a heuristic algorithm to support schema integration. First, the attributes of every class are partitioned in such a way that the isolation ratio of every element in the partition is one, and every class is factorised by delegation if desirable. Subsequently, for every pair of promising classes, a set of possible superclasses is computed, and both classes are factorised by specialisation or redefined as a relation if desirable.

Algorithm 11.14 The following algorithm is a heuristic for integrating two database schemas (resp., DBS1 and DBS2), given thresholds for strong relatedness and weak relatedness (resp., TSR and TWR):

```

integrate(DBS1,DBS2,TSR,TWR) =
  for every class C in DBS1 or DBS2
    do for every element A in partition(C)
      do if  $strongrel(name(C),A) \geq TSR$  and  $1 < |A| < |atts(C)|$ 
        then create class C1 as the class containing A
          and the methods that refer to A;
          factorise C by delegation using C1;
          mark C and C1
        elif  $weakrel(name(C),A) \geq TWR$ 
        then mark C
      fi
    od
  od;
  for every marked C1 in DBS1
    do for every marked C2 in DBS2
      do if there is a superclass C of a class in  $joins(C1,C2)$  that
        can be used as a factor according to the designer
        then transform(C1,C2,C)
      fi
    od
  od
;
transform(C1,C2,C) =
  begin let  $f_1$  be an injection from  $atts(C)$  to  $atts(C1)$  induced by  $C1 \preceq C$ ;
    let  $f_2$  be an injection from  $atts(C)$  to  $atts(C2)$  induced by  $C2 \preceq C$ ;
    define A1 as the attribute names in the range of  $f_1$ ;
    define A2 as the attribute names in the range of  $f_2$ ;
    if  $isolation(name(C1),A1) < 1$  or  $isolation(name(C2),A2) < 1$ 

```

```

then factorise C1 and C2 by specialisation using C
elif  $1 < |A1| < |atts(C1)|$  and  $1 < |A2| < |atts(C2)|$ 
then factorise C1 and C2 or redefine C1 and C2 as relations
      according to the choice of the designer
else factorise C1 and C2 by specialisation using C
fi
end

```

where $partition(C)$ is constructed as follows:

```

 $graph(C)$  has a node for every attribute name in  $atts(C)$ 
 $graph(C)$  has an edge between two nodes if there is a method in the set of
  all methods of  $C$  in which both attribute names occur
 $partition(C)$  consists of sets of attribute names,
  one set for every connected subgraph of  $graph(C)$ :
  two attribute names are in the same set if their nodes are connected
  two attribute names are in different sets if their nodes are not connected.

```

□

Note that the algorithm interacts with the designer. It should be mentioned again that the algorithm is a heuristic and should therefore be used in close interaction with the designer. The heuristic can be improved by combining the different thresholds and refining the different actions. We conclude this chapter with an example.

Example 11.15 The following class hierarchy is a part of drawing tool A:

```

Class Square
Attributes
  pos:Pos
  width:integer
Methods
  set (x:integer, y:integer) =
    pos := pos.set_pos(x,y)
  translate (dx:integer, dy:integer) =
    pos := pos.translate_pos(dx,dy)
Endclass
Class Pos
Attributes
  x_co:integer
  y_co:integer
Methods
  set_pos (x:integer, y:integer → p:Pos) =
    p := new(Pos, x_co=x, y_co=y)

```

```

    translate_pos (dx:integer, dy:integer → p:Pos) =
      p := new(Pos, x_co=x_co+dx, y_co=y_co+dy)
  Endclass.

```

Objects in class Square have a position on the screen and a width, and can be moved around on the screen using method set and method translate. The following class hierarchy is a part of drawing tool B:

Class Rectangle

Attributes

```

    pos_x:integer
    pos_y:integer
    width_x:integer
    width_y:integer

```

Methods

```

    put (x:integer, y:integer) =
      pos_x := x; pos_y := y
    move (delta_x:integer, delta_y:integer) =
      pos_x := pos_x + delta_x; pos_y := pos_y + delta_y

```

Endclass.

Objects in class Rectangle have a position w.r.t the x-axis, a position w.r.t. the y-axis, a length, and a width, and can be moved around on the screen using method put and method move.

Let C_S be class Square, C_P be class Pos, and C_R be class Rectangle. The partitions of these classes are given by:

```

partition( $C_S$ ) = {{pos}, {width}}
partition( $C_P$ ) = {{x_co, y_co}}
partition( $C_R$ ) = {{pos_x, pos_y}, {width_x}, {width_y}}.

```

Now, let us use Algorithm 11.6 to integrate the class hierarchies. In the first big loop, all classes are marked and Rectangle is factorised using delegation:

Class Rectangle

Attributes

```

    pos:X
    width_x:integer
    width_y:integer

```

Methods

```

    put (x:integer, y:integer) =
      pos := pos.put_pos (x,y)
    move (dx:integer, dy:integer) =
      pos := pos.move_pos (dx,dy)

```

Endclass


```

Class X
Attributes
  pos_x:integer
  pos_y:integer
Methods
  put_pos (x:integer, y:integer → l:X) =
    l := new(X, pos_x=x, pos_y=y)
  move_pos (dx:integer, dy:integer → l:X) =
    l := new(X, pos_x=pos_x+dx, pos_y=pos_y+dy)
Endclass.

```

Let C'_R be the new class Rectangle and C_X be class X. In the second big loop, the only interesting join sets are:

$$\begin{aligned}
 joins(C_P, C_X) &= \{\gamma_P(C_P)\} = \{\gamma_X(C_X)\} \\
 joins(C_S, C'_R) &= \{\gamma_S(C_S)\},
 \end{aligned}$$

where the γ 's are rename functions replacing attribute names and method names.

Suppose the designer chooses C_P as a factor to factorise C_P and C_X (redefinition as a relation is not possible). Since C_P and C_X are equivalent, it suffices to remove C_X and redefine every class D that refers to C_X by replacing the occurrences of X in D by Pos and redefine every method in D in which an attribute or method name *name* occurs that refers to an attribute or method of C_X by replacing these occurrences of *name* by the corresponding attribute or method name in Pos. Let C''_R be the redefined class Rectangle and suppose the designer chooses the following class as a factor to factorise C_S and C''_R (again, redefinition as a relation is not possible):

```

Class Figure
Attributes
  pos:Pos
Methods
  set (x:integer, y:integer) =
    pos := pos.set_pos (x,y)
  translate (dx:integer, dy:integer) =
    pos := pos.translate_pos (dx,dy)
Endclass.

```

Then the integrated class hierarchy will consist of class Pos, class Figure, and:

```

Class Square Isa Figure
Attributes
  width:integer
Endclass
Class Rectangle Isa Figure

```

Attributes

width_x:integer

width_y:integer

Endclass.

Note that the only real choice made by the designer is the choice to factorise Square and Rectangle using Figure instead of Square. \square

Chapter 12

Conclusion

In this final chapter, we summarise and evaluate the theory developed in Part I and Part II of this thesis. Furthermore, we discuss some directions for further research.

12.1 Summary

In Part I, a theory for comparing and integrating database schemas has been developed in the context of an object-oriented data model. An object-oriented data model has been chosen, because such a data model offers an integrated formalism for defining both structure and behaviour.

In Chapter 3, object-oriented database schemas have been introduced and formalised in terms of types and functions. An object-oriented database schema consists of classes related by a subclass relation. A class defines the structure of its instances in terms of attributes and the behaviour of its instances in terms of methods. Redefinition of attributes and methods in subclasses is restricted to specialisation. The reason for this restriction is that specialisation preserves the semantics of inherited attributes and inherited methods in the subclass. Furthermore, an instance at the level of a subclass hides the fact that it is specialised at the level of the superclass.

Methods are used to query and modify instances and can be used by other methods. In this way, methods define an interface between instances of different classes and between instances of the same class. We have introduced a restricted interface, where an instance is allowed to query and modify itself and to query, but not modify, other instances. If an object wants to modify an attribute that refers to another object, it can only modify the reference. As a result, the former object refers to yet another object and the latter object remains unchanged. The restricted interface simplifies method comparison, which is an important aspect

of our approach towards schema integration.

The formalisation in terms of types and functions extends the approach developed in TM/FM [4] with recursive types. In TM/FM, object identifiers are used to distinguish between different instances and to cope with recursive classes and references to other classes. As a result, only a part of the structure of a class is represented by its type; the other part is represented by a constraint. Our formalisation uses object identifiers to distinguish between different instances and a recursion operator in combination with type variables to cope with recursive classes and references to other classes. The advantage is that the complete structure of a class is represented by its type, which makes attribute comparison easier.

In Chapter 4, two notions of semantic equivalence and two notions of semantic subtyping have been introduced. Structural equivalence and structural subtyping are defined in terms of trees, which give the relationship between underlying types and schemas in the graph-based data model GOOD [23]. In fact, our trees can be obtained from the graphs of GOOD by unfolding them to remove cycles and by adding object identifiers. Extensional equivalence and extensional subtyping are defined in terms of the extension of a type, i.e., the set of its instances.

Equivalence of instances is also defined in terms of trees, which give the relationship between instances of underlying types and instances of schemas in GOOD. Since types and instances are very similar, we have the following: whenever two types are equivalent, then so are their extensions. This means that if two schemas are integrated using type equivalence, their instances can be integrated as well.

Furthermore, derivation systems for type equivalence and subtyping have been introduced, which are decidable and sound and complete with respect to structural and extensional semantics.

In Chapter 5, equality of functional forms has been defined and proven decidable by means of an algorithm. This algorithm can be used to define a derivation system for equality of functional forms. Furthermore, a subfunction relation has been defined and proven decidable.

In Chapters 6 and 7, the subtype relation and the subfunction relation have been combined into a subclass relation. The subclass relation compares classes by the structure of their instances (subtype relation) and by the way these instances are manipulated (subfunction relation). That is, classes are semantically the same if their instances have the same structure and are manipulated in the same way. The combination of structure and behaviour can be regarded as abstract semantics, i.e., as an approximation to real world semantics, because behaviour is a very important aspect of real world objects.

Furthermore, the subclass relation has been extended based on morphisms between classes and semantic properties of methods. The (extended) subclass relation induces a lattice structure on classes with a meet and a join operator. The join operator, which defines the most specialised common superclass, can be

used to factorise classes and integrate class hierarchies. Hence, schema integration is based on the abstract semantics of the subclass relation.

In Part II, the theory developed in Part I has been extended with schema transformations and syntactic properties of methods. To cope with transformations of methods, the data model introduced in Chapter 3 has been extended with query methods, method calls, and object creation in Chapter 8.

Furthermore, to generalise the approach, the data model has been extended with multiple inheritance for attributes and with keys, as well as the formalisation of the data model in terms of types, predicates, and functions. Multiple inheritance of attributes is restricted in such a way that the type of an attribute in the subclass is a subtype of any of the corresponding types in the superclasses. The reason for this restriction is again to preserve the semantics of inherited attributes in the subclass.

In Chapters 9 and 10, a number of type transformations have been introduced, a subset of which is sound and complete with respect to data capacity. The type transformations induce transformations on predicates and functions, which can be combined into transformations on classes. Since types and instances are very similar, we have the following: whenever two types are equivalent after transformation, then so are their instances. This means that if two schemas are integrated using type transformations, their instances can be integrated as well.

Finally, in Chapter 11, the subclass morphisms of Chapters 6 and 7 have been adapted to cope with keys and query methods. Again, the adapted subclass morphisms induce a lattice structure on classes with a meet and a join operator. Furthermore, schema comparison and schema integration have been extended with transformations on classes and syntactic properties of methods. Class transformations are used to detect transformational similarities between classes. As a result, more (and more complex) similarities can be detected before interaction with the designer. Syntactic properties of methods are used to guide the detection of subclass morphisms. As a result, the least promising subclass morphisms can be eliminated before method comparison and, hence, before interaction with the designer.

12.2 Evaluation

The goal of this thesis is to integrate object-oriented database schemas. To integrate schemas, we have to compare them first. Since it is not possible to use real world semantics, one can try to compare schemas using a (partial) approximation to real world semantics.

In this thesis, we have compared schemas using structure and behaviour, because we think that behaviour is a very important aspect of real world objects. The resulting approach towards schema integration has the following properties:

1. it has a sound mathematical basis
2. it is an extension to existing approaches
3. it reduces the amount of work for the designer
4. it can be implemented in a tool.

Our approach is an extension to existing approaches, because existing approaches do not use behaviour at all, nor do they use transformations to detect similarities between schemas. The use of behaviour makes our approach more complete in a semantic sense and reduces the amount of work for the designer. The use of transformations makes it possible to detect more complex similarities between schemas. Our approach can be implemented in a tool, because it has a formal basis and all properties are decidable.

Finally, although the amount of work for the designer has been reduced, the comparison algorithm itself has high complexity. To overcome this problem, we have introduced syntactic properties of methods to decrease the number of method comparisons.

12.3 Further research

The first direction for further research is generalisation and improvement of the theoretical framework. Possible generalisations and improvements are:

1. extension of the data model
2. refinement of the abstract semantics
3. reduction of the computational complexity.

The data model can be extended with new types, such as discriminated union types and function types, with new static constraints, based on a first order language, with new methods, such as O₂-like update and query methods [34], and with dynamic constraints, based on process algebra. To cope with these extensions, the subclass relation introduced in Chapter 11 has to be generalised. Fortunately, the join operator for the generalised subclass relation is obtained in exactly the same way as for the original subclass relation. This means that our approach towards schema integration is still applicable. However, since equivalence of general constraints and general methods is undecidable, we have to use a heuristic approach to compare constraints and methods. The heuristic approach can be based on sufficient conditions or on probabilistic equivalence relations.

Furthermore, the data model can be extended with multiple inheritance for methods and with a database level on top of the class definitions. The database

level contains class definitions, but also database constraints and database methods, similar to TM/FM. A database constraint is a constraint on objects of different classes and a database method is a method that manipulates objects of different classes. Since constraints at the class level can only define constraints on objects of the same class and methods on the class level can only define methods that manipulate objects of the same class, the database level is required in order to define complex applications.

The abstract semantics of class hierarchies can be refined by extending the set of characteristics used for schema comparison; for example, linguistic information or information from a data dictionary. Furthermore, the abstract semantics of class hierarchies can be refined by extending the set of schema transformations in order to obtain new schema equivalences.

The computational complexity can be reduced by using a heuristic approach to decrease the number of method comparisons and to reduce the complexity of one method comparison. The number of comparisons can be reduced by using properties of methods and the complexity of a comparison can be reduced by using sufficient conditions or probabilistic equivalence relations.

The second direction for further research is a practical validation of the theoretical framework (theoretical validation of the framework has been done in the previous section). Practical validation should be done for real world applications, based on the schema integration tool of the previous section.

Appendix A

Complexity of subclass relation

First, we show that testing whether two classes have a subclass relationship is non-deterministic polynomial time decidable (NP [27]).

Let C_1 and C_2 be classes in a well-defined class hierarchy. Furthermore, let the formalisation of C_1 and C_2 be given by:

$$\begin{aligned} (type(C_1), func(C_1)) &= (\tau_1, \{\lambda obj:\tau_1 \lambda P_1.f_1, \dots, \lambda obj:\tau_1 \lambda P_n.f_n\}) \\ (type(C_2), func(C_2)) &= (\tau_2, \{\lambda obj:\tau_2 \lambda Q_1.g_1, \dots, \lambda obj:\tau_2 \lambda Q_m.g_m\}). \end{aligned}$$

Now, C_1 is a subclass of C_2 if and only if there exists a subtype morphism φ from $type(C_2)$ to $type(C_1)$ and a function ψ from $\{1, \dots, m\}$ to $\{1, \dots, n\}$ and, for every $i \in \{1, \dots, m\}$, a permutation \tilde{Q}_i of Q_i , such that:

$$a) \forall i \in \{1, \dots, m\} [gen(f_{\psi(i)}, \tau_2) =_{FUNC} \varphi(\lambda obj:\tau_2 \lambda \tilde{Q}_i.g_i)].$$

We can construct a witness for testing whether C_1 is a subclass of C_2 as follows: construct a type morphism φ from $type(C_1)$ to $type(C_2)$, a function ψ from $\{1, \dots, m\}$ to $\{1, \dots, n\}$, and a permutation \tilde{Q}_i for every $i \in \{1, \dots, m\}$. The witness can be constructed in polynomial time and then a) can be tested in polynomial time using the witness.

Second, we show that testing whether a graph has a Hamilton cycle ([27]) can be encoded as testing whether two classes have a subclass relationship.

Let $G = (N, E)$ be a graph, where $N = \{x_1, \dots, x_n\}$ is the set of nodes and $E = \{(v_1, w_1), \dots, (v_p, w_p)\} \subseteq N \times N$ is the set of edges. A cycle in G is a sequence of nodes $(x'_1, x'_2, \dots, x'_q)$, such that:

1. $(x'_i, x'_{i+1}) \in E$ for every $i \in \{1, \dots, q-1\}$
2. $(x'_q, x'_1) \in E$.

A cycle in G is a Hamilton cycle if and only if every node of G occurs exactly once in the cycle. Hamilton cycle $(x'_1, x'_2, \dots, x'_n)$ in G corresponds to a Hamiltonian subgraph of G , viz., $(N, \{(x'_1, x'_2), \dots, (x'_{n-1}, x'_n), (x'_n, x'_1)\})$.

Testing whether graph G has a Hamilton cycle can be done by considering every possible permutation of the nodes of G and testing whether it is a Hamilton cycle. This can be encoded as follows. Let a_i be a unique name for node x_i (for every $i \in \{1, \dots, n\}$). Graph G is encoded as a class, of which the attributes correspond to the nodes and the methods to the edges:

```

Class Graph
Attributes
   $a$  : string
   $a_1$  : string
   $a_2$  : string
   $\vdots$ 
   $a_n$  : string
Methods
   $m_1() = a := b_1 + c_1$ 
   $m_2() = a := b_2 + c_2$ 
   $\vdots$ 
   $m_p() = a := b_p + c_p$ 
Endclass

```

where b_i is the name corresponding to node v_i and c_i is the name corresponding to node w_i . Note that transforming a graph into a class is a polynomial time transformation.

Hamiltonian graph $HG = (N, \{(x_1, x_2), \dots, (x_{n-1}, x_n), (x_n, x_1)\})$ is encoded in the same way as G :

```

Class Hamiltonian.Graph
Attributes
   $a$  : string
   $a_1$  : string
   $a_2$  : string
   $\vdots$ 
   $a_n$  : string
Methods
   $m_1() = a := a_1 + a_2$ 
   $m_2() = a := a_2 + a_3$ 
   $\vdots$ 
   $m_{n-1}() = a := a_{n-1} + a_n$ 

```

$$m_n() = a := a_n + a_1$$

Endclass.

Let C_G be class 'Graph', C_{HG} be class 'Hamiltonian_Graph', τ be $type(C_G)$, φ be the identity type morphism on τ , $\{f_1, \dots, f_p\}$ be $funcs(C_G)$, and $\{g_1, \dots, g_n\}$ be $funcs(C_{HG})$.

Suppose $h = (x'_1, x'_2, \dots, x'_n)$ is a Hamilton cycle in G . Then there is a subtype morphism φ from τ to τ and a function ψ from $\{1, \dots, n\}$ to $\{1, \dots, p\}$, such that:

$$b) \forall i \in \{1, \dots, n\} [f_{\psi(i)} \leq_{FUNC}^w \varphi(g_i)],$$

which means that C_G is a subclass of C_{HG} . Subtype morphism φ encodes h as a permutation of the nodes of G and requirement b encodes the fact that every edge in h must occur as an edge in graph G .

Suppose C_G is a subclass of C_{HG} . Then there is a subtype morphism φ from τ to τ and a function ψ from $\{1, \dots, n\}$ to $\{1, \dots, p\}$, such that:

$$\forall i \in \{1, \dots, n\} [f_{\psi(i)} \leq_{FUNC}^w \varphi(g_i)].$$

Then $(x'_1, x'_2, \dots, x'_n)$, where $x'_i = x_j$ if $\varphi(a_i) = a_j$, is a Hamilton cycle in G .

To summarise, testing for a subclass relationship is NP and the NP-complete problem of testing for a Hamilton cycle can be encoded as testing for a subclass relationship. Hence, we can conclude that testing for a subclass relationship is NP-complete as well.

Bibliography

- [1] S. Abiteboul and R. Hull. Restructuring hierarchical database objects. *Theoretical Computer Science*, 62:3–38, 1988.
- [2] A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly typed, interactive conceptual language. *ACM Trans. on Database Systems*, 10(2):230–260, 1985.
- [3] R. Amadio and L. Cardelli. Subtyping recursive types. In *Proc. Int. Symp. on Principles of Programming Languages*, pages 104–118, 1991.
- [4] P. Apers, H. Balsters, R. de By, and C. de Vreeze. Inheritance in an object-oriented data model. Memoranda Informatica 90-77, University of Twente, Enschede, The Netherlands, 1990.
- [5] H. Balsters, R. de By, and R. Zicari. Typed sets as a basis for object-oriented database schemas. In *Proc. Computing Science in the Netherlands*, pages 62–77. Stichting Mathematisch Centrum, Amsterdam, The Netherlands, 1991.
- [6] H. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, volume 2*, pages 118–309. University Press, New York, NY, 1992.
- [7] C. Batini and M. Lenzerini. A methodology for data schema integration in the ER model. *IEEE Transactions on Software Engineering*, pages 650–664, November 1984.
- [8] C. Batini, M. Lenzerini, and S. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.
- [9] J. Biskup and B. Convent. A formal view integration method. In *Proc. Int. Conference on Management of Data*, pages 398–407, 1986.
- [10] M. Bouzeghoub and I. Comyn-Wattiau. View integration by semantic unification and transformation of data structures. In *Proc. Int. Conference on the Entity-Relationship Approach*, pages 381–398. North-Holland, 1990.

- [11] M. Bright and A. Hurson. Linguistic support for semantic identification and interpretation in multidatabases. In *Proc. Int. Workshop on Interoperability in Multidatabase Systems*, pages 306–313, 1991.
- [12] P. Buneman, S. Davidson, and A. Kosky. Theoretical aspects of schema merging. In *Proc. European Conference on Database Technology (EDBT)*, pages 152–167, 1992.
- [13] L. Cardelli. A semantics of multiple inheritance. In *Proc. Int. Symp. on Semantics of Datatypes, LNCS 173*, pages 51–67. Springer-Verlag, Berlin, 1984.
- [14] M. Casanova and V. Vidal. Towards a sound view integration methodology. In *Proc. Symposium on Principles of Database Systems*, pages 36–47, 1983.
- [15] U. Dayal and H. Hwang. View definition and generalization for database integration in a multidatabase system. *IEEE Trans. on Software Engineering*, 10(6):628–644, 1984.
- [16] J. de Souza. SIS – A schema integration system. In *Proc. British National Conference on Databases*, pages 167–185, 1986.
- [17] C. de Vreeze. Formalization of inheritance of methods in an object-oriented data model. Memoranda Informatica 90-76, University of Twente, Enschede, The Netherlands, 1990.
- [18] R. Elmasri and S. Navathe. Object integration in logical database design. In *Proc. Int. Conference on Data Engineering*, pages 426–433, 1984.
- [19] R. Elmasri and G. Wiederhold. Data model integration using the structural model. In *Proc. Int. Conf. on Management of Data*, pages 191–202, 1979.
- [20] P. Fankhauser, M. Kracker, and E. Neuhold. Semantic vs. structural resemblance of classes. *ACM SIGMOD Record*, 20(4):59–63, 1991.
- [21] D. Gangopadhyay and T. Barsalou. On the semantic equivalence of heterogeneous representations in multimodel multidatabase systems. *SIGMOD Record*, 20(4):35–39, 1991.
- [22] M. Garcia-Solaco, M. Castellanos, and F. Saltor. Discovering interdatabase resemblance of classes for interoperable databases. In *Proc. Research Issues in Data Engineering – Interoperability in Multidatabase Systems (RIDE-IMS)*, 1993.
- [23] M. Gyssens, J. Paredaens, and D. van Gucht. A graph-oriented object database model. In *Proc. Int. Symp. on Principles of Database Systems*, pages 417–424, 1990.

- [24] J.-L. Hainaut. Entity-generating schema transformations for entity-relationship models. In *Proc. Int. Conf. on the Entity-Relationship Approach*, 1991.
- [25] S. Hayne and S. Ram. Multi-user view integration system (MUVIS): An expert system for view integration. In *Proc. Int. Conference on Data Engineering*, pages 402–409, 1990.
- [26] D. Heimbigner and D. McLeod. A federated architecture for information management. *ACM Trans. on Office Information Systems*, pages 253–278, 1985.
- [27] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, 1979.
- [28] P. Johannesson. Schema transformations as an aid in view integration. In *Proc. Int. Conf. on Advanced Information Systems Engineering, LNCS 685*, pages 71–92. Springer-Verlag, Berlin, 1993.
- [29] M. Kersten. Goblin: a DBPL designed for advanced database applications. In *Proc. Int. Conf. on Database and Expert Systems Applications*, pages 345–349. Springer-Verlag, Wien, 1991.
- [30] M. Kersten, C. van den Berg, A. Siebes, C. Thieme, and M. van der Voort. The Goblin database programming language. Report CS-R9407, CWI, Amsterdam, The Netherlands, 1994 (available as pub/CWIreports/AA/CS-R9407.ps.Z from ftp.cwi.nl).
- [31] J.-W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science, volume 2*, pages 1–116. University Press, New York, NY, 1992.
- [32] C. Koster. On infinite modes. *ACM SIGPLAN Notices*, 4(3):109–112, 1969.
- [33] J. Larson, S. Navathe, and R. Elmasri. A theory of attribute equivalence in databases with application to schema integration. *IEEE Transactions on Software Engineering*, pages 449–463, 1989.
- [34] C. Lécluse and P. Richard. The O₂ database programming language. In *Proc. Int. Conf. on Very Large Databases*, pages 411–422. Morgan Kaufmann, Palo Alto, CA, 1989.
- [35] A. Macintyre. The laws of exponentiation. In *Model Theory and Arithmetic, LNM 890*, pages 185–197. Springer-Verlag, Berlin, 1979.
- [36] M. Mannino and W. Effelsberg. Matching techniques in global schema design. In *Proc. Int. Conference on Data Engineering (COMPDEC)*, pages 418–425, 1984.

- [37] M. Mannino, S. Navathe, and W. Effelsberg. A rule based approach for merging generalisation hierarchies. *Information Systems*, 13(3):257–272, 1988.
- [38] A. Motro. Superviews: Virtual integration of multiple databases. *IEEE Trans. on Software Engineering*, 13(7):785–798, 1987.
- [39] A. Motro and P. Buneman. Constructing superviews. In *Proc. Int. Conf. on Management of Data*, pages 56–64, 1981.
- [40] S. Navathe, R. Elmasri, and J. Larson. Integrating user views in database design. *IEEE Computer*, pages 50–62, 1986.
- [41] S. Navathe and S. Gadgil. A methodology for view integration in logical data base design. In *Proc. Int. Conf. on Very Large Databases*, pages 142–155, 1982.
- [42] S. Navathe, T. Sashidhar, and R. Elmasri. Relationship merging in schema integration. In *Proc. Int. Conference on Very Large Databases*, pages 78–90, 1984.
- [43] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.
- [44] E. Rundensteiner. Multiview: A methodology for supporting multiple views in object-oriented databases. In *Proc. Int. Conference on Very Large Databases*, pages 187–198, 1992.
- [45] F. Saltor, M. Castellanos, and M. Garcia-Solaco. Overcoming schematic discrepancies in interoperable databases. In *Proc. Conference on Interoperable Database Systems (DS-5)*, pages 191–205, 1992.
- [46] A. Sheth and S. Gala. Attribute relationships: an impediment in automating schema integration. In *Proc. Workshop on Heterogeneous Database Systems*, 1989.
- [47] A. Sheth, S. Gala, and S. Navathe. On automatic reasoning for schema integration. *Int. Journal of Intelligent and Cooperative Information Systems*, 2(1):23–50, 1993.
- [48] A. Sheth and J. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 20(3):183–236, 1990.
- [49] A. Sheth, J. Larson, A. Cornelio, and S. Navathe. A tool for integrating conceptual schemas and user views. In *Proc. Int. Conference on Data Engineering*, pages 176–183, 1988.

- [50] A. Sheth and H. Marcus. Schema analysis and integration: Methodology, techniques, and prototype toolkit. Technical memorandum TM-ST-019981/1, Bell Communications Research, Piscataway, NJ 08854, 1992.
- [51] D. Shipman. The functional data model and the data language DAPLEX. In *Readings in Object-Oriented Database Systems*, pages 95–111. 1990.
- [52] P. Shoval and S. Zohn. Binary-relationship integration methodology. *Data & Knowledge Engineering*, 6:225–250, 1991.
- [53] M. Siegel and S. Madnick. A metadata approach to resolving semantic conflicts. In *Proc. International Conference on Very Large Databases*, pages 133–145, 1991.
- [54] S. Spaccapietra and C. Parent. Conflicts and correspondence assertions in interoperable databases. *SIGMOD Record*, 20(4):49–54, 1991.
- [55] S. Spaccapietra and C. Parent. View integration: A step forward in solving structural conflicts. *IEEE Trans. on Knowledge and Data Engineering*, 6(2):258–274, 1994.
- [56] S. Spaccapietra, C. Parent, and Y. Dupont. Model independent assertions for integration of heterogeneous schemas. *VLDB Journal*, 1:81–126, 1992.
- [57] C. Thieme and A. Siebes. Schema integration in object-oriented databases. In *Proc. Int. Conf. on Advanced Information Systems Engineering, LNCS 685*, pages 54–70. Springer-Verlag, Berlin, 1993. An extended version is available as `pub/CWIreports/AA/CS-R9320.ps.Z` from `ftp.cwi.nl`.
- [58] C. Thieme and A. Siebes. Schema refinement and schema integration in object-oriented databases. In *Proc. Computing Science in The Netherlands, ISBN 90 6196 430 X*, pages 343–354. Stichting Mathematisch Centrum, 1993. An extended version is available as `pub/CWIreports/AA/CS-R9354.ps.Z` from `ftp.cwi.nl`.
- [59] C. Thieme and A. Siebes. An approach to schema integration based on transformations and behaviour. In *Proc. Int. Conf. on Advanced Information Systems Engineering, LNCS 811*, pages 297–310. Springer-Verlag, Berlin, 1994. An extended version is available as `pub/CWIreports/AA/CS-R9403.ps.Z` from `ftp.cwi.nl`.
- [60] C. Thieme and A. Siebes. Type equivalence, subtyping, and type transformations in object-oriented databases. Report CS-R9451, CWI, Amsterdam, The Netherlands, 1994 (available as `pub/CWIreports/AA/CS-R9451.ps.Z` from `ftp.cwi.nl`).

- [61] C. Thieme and A. Siebes. Guiding schema integration by behavioural information. *Information Systems*, 1995. To be published.
- [62] J. Ullman. *Principles of Database and Knowledge Base Systems, volume 1*. Computer Science Press, Rockville, Maryland, 1988.
- [63] S. Urban and J. Wu. Resolving semantic heterogeneity through the explicit representation of data model semantics. *SIGMOD Record*, 20(4):55–58, 1991.
- [64] C. Yu, B. Jia, W. Sun, and S. Dao. Determining relationships among names in heterogeneous databases. *SIGMOD Record*, 20(4):79–80, 1991.
- [65] C. Yu, W. Sun, S. Dao, and D. Keirse. Determining relationships among attributes for interoperability of multi-database systems. In *Proc. Int. Workshop on Interoperability in Multidatabase Systems*, pages 251–257, 1991.

Samenvatting

Samenwerking tussen informatiesystemen wordt steeds belangrijker, omdat organisaties en delen van organisaties steeds meer (moeten) samenwerken. Denk bijvoorbeeld aan de fusie van twee banken, waarbij de transactiesystemen gecombineerd moeten worden, of aan samenwerking tussen nationale politiemachten, waarbij informatie uitgewisseld moet worden.

Voor samenwerking tussen informatiesystemen is het nodig dat gegevensbestanden gekoppeld kunnen worden. Gegevensbestanden representeren objecten in de 'echte wereld', zoals personen en auto's. Koppeling kan op verschillende manieren gerealiseerd worden, afhankelijk van factoren zoals heterogeniteit en autonomie van de subsystemen. Welke keuze er ook gemaakt wordt, er zal altijd schema integratie aan te pas komen.

Een schema beschrijft hoe objecten er uit kunnen zien (structuur) en hoe objecten kunnen veranderen (gedrag). Schema integratie is het proces om een aantal schema's te verenigen tot een nieuw schema dat correct, minimaal en begrijpelijk is. Minimaliteit eist dat we kunnen detecteren of twee delen van verschillende schema's hetzelfde concept representeren volgens de semantiek van de 'echte wereld'. Dit maakt schema integratie tot een zeer moeilijk proces. In het algemeen bestaat schema integratie uit vier fasen, waarvan vergelijking en aanpassing de twee belangrijkste zijn. In de vergelijkingsfase worden schema's geanalyseerd om overeenkomsten en conflicten tussen schema's te detecteren. In de aanpassingsfase moeten de conflicten uit de tweede stap worden opgelost, onder andere met behulp van schema transformaties.

Bestaande methoden voor schema integratie kijken naar structuur en niet naar gedrag. Dit proefschrift beschrijft een methode die op een aantal punten verschilt van de bestaande. Zo worden er al in de vergelijkingsfase schema transformaties gebruikt om overeenkomsten tussen objecten te bepalen. Ook wordt er naar gedrag gekeken om overeenkomsten tussen objecten te detecteren. In het eerste deel worden object-georiënteerde schema's geïntroduceerd en vertaald naar formele typen en functies. Ook wordt beschreven hoe typen, die structuur representeren, geïntegreerd kunnen worden en hoe functies, die gedrag representeren, geïntegreerd kunnen worden. In het tweede deel worden de schema's uit het eerste

deel uitgebreid en de vertaling naar formele typen en functies aangepast. Verder worden integratie van typen en integratie van functies uitgebreid met behulp van schema transformaties.

Voordelen van de in dit proefschrift beschreven methode zijn: zij heeft een formele basis, is een uitbreiding op bestaande methoden, reduceert de hoeveelheid werk voor de gebruiker en kan geïmplementeerd worden in een tool.