Domain-Specific Languages for Enterprise Systems

Tom Hvitved Patrick Bahr Jesper Andersen

Department of Computer Science, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen, Denmark
{hvitved,bahr,jespera}@diku.dk

Abstract

The process-oriented event-driven transaction systems (POETS) architecture introduced by Henglein et al. is a novel software architecture for enterprise resource planning (ERP) systems. POETS employs a pragmatic separation between (i) transactional data, that is what has happened; (ii) reports, that is what can be derived from the transactional data; and (iii) contracts, that is which transactions are expected in the future. Moreover, POETS applies domain-specific languages (DSLs) for specifying reports and contracts, in order to enable succinct declarative specifications as well as rapid adaptability and customisation. In this report we document an implementation of a generalised and extended variant of the POETS architecture. The generalisation is manifested in a detachment from the ERP domain, which is rather an instantiation of the system than a built-in assumption. The extensions amount to a customisable data model based on nominal subtyping; support for run-time changes to the data model, reports and contracts, while retaining full auditability; and support for referable data that may evolve over time, also while retaining full auditability as well as referential integrity. Besides the revised architecture, we present the DSLs used to specify data definitions, reports, and contracts respectively, and we provide the complete specification for a use case scenario, which demonstrates the conciseness and validity of our approach. Lastly, we describe technical aspects of our implementation, with focus on the techniques used to implement the tightly coupled DSLs.

Contents

1 Introduction

Enterprise resource planning (ERP) systems are comprehensive software systems used to manage daily activities in enterprises. Such activities include—but are not limited to—financial management (accounting), production planning, supply chain management and customer relationship management. ERP systems emerged as a remedy to heterogeneous systems, in which data and functionality are spread out—and duplicated—amongst dedicated subsystems. Instead, an ERP system it built around a central database, which stores all information in one place.

Traditional ERP systems such as Microsoft Dynamics NAV[1], Microsoft Dynamics AX[2], and SAP[3] are three-tier architectures with a client, an application server, and a centralised relational database system. The central database stores information in tables, and the application server provides the business logic, typically coded in a general purpose, imperative programming language. A shortcoming to this approach is that the state of the system is decoupled from the business logic, which means that business processes—that is, the daily activities—are not represented explicitly in the system. Rather, business processes are encoded implicitly as transition systems, where the state is maintained by tables in the database, and transitions are encoded in the application server, possibly spread out across several different logical modules.

The process-oriented event-driven transaction systems (POETS) architecture introduced by Henglein et al. [6] is a qualitatively different approach to ERP systems. Rather than storing both transactional data and implicit process state in a database, POETS employs a pragmatic separation between transactional data, which is persisted in an *event log*, and *contracts*, which are explicit representations of business processes, stored in a separate module. Moreover, rather than using general purpose programming languages to specify business processes, POETS utilises a declarative domain-specific language (DSL) [1]. The use of a DSL not only enables explicit formalisation of business processes, it also

[1]http://www.microsoft.com/en-us/dynamics/products/nav-overview.aspx.
[2]http://www.microsoft.com/en-us/dynamics/products/ax-overview.aspx.
[3]http://www.sap.com.

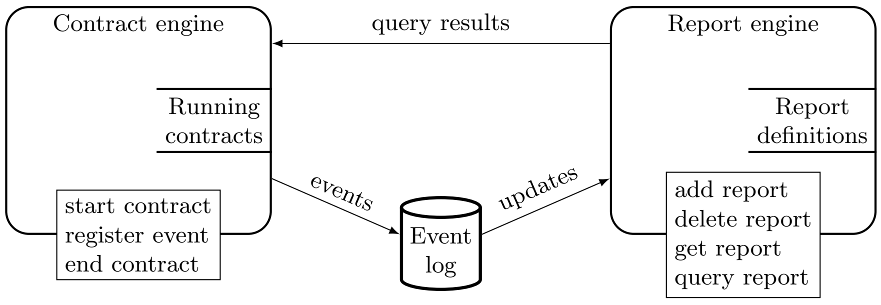

Figure 1: Bird's-eye view of the POETS architecture (diagram copied from [6]).

minimises the gap between requirements and a running system. In fact, Henglein et al. take it as a goal of POETS that "[...] the formalized requirements *are* the system" [6, page 382].

The bird's-eye view of the POETS architecture is presented in Figure 1. At the heart of the system is the event log, which is an append-only list of transactions. Transactions represent "things that take place" such as a payment by a customer, a delivery of goods by a shipping agency, or a movement of items in an inventory. The append-only restriction serves two purposes. First, it is a legal requirement in ERP systems that transactions, which are relevant for auditing, are retained. Second, the report engine utilises monotonicity of the event log for optimisation, as shown by Nissen and Larsen [12].

Whereas the event log stores historical data, contracts play the role of describing which events are expected in the future. For instance, a yearly payment of value-added tax (VAT) to the tax authorities is an example of a (recurring) business process. The amount to be paid to the tax authorities depends, of course, on the financial transactions that have taken place. Therefore, information has to be derived from previous transactions in the event log, which is realised as a *report*. A report provides structured data derived from the transactions in the event log. Like contracts, reports are written in a declarative domain-specific language—not only in order to minimise the semantic gap between requirements and the running system, but also in order to perform automatic optimisations.

Besides the radically different software architecture, POETS distinguishes itself from existing ERP systems by abandoning the double-entry bookkeeping (DEB) accounting principle [17] in favour of the resources, events, and agents (REA) accounting model of McCarthy [10].

In double-entry bookkeeping, each transaction is recorded as two postings in a *ledger*—a *debit* and a *credit*. When, for instance, a customer pays an amount x to a company, then a debit of x is posted in a cash account, and a credit of x is posted in a sales account, which reflects the flow of cash from the customer to the company. The central invariant of DEB is that the total credit equals the total debit—if not, resources have either vanished or spontaneously appeared. DEB fits naturally in the relational database oriented architectures, since each ledger is similar in structure to a table. Moreover, DEB is the de facto standard accounting method, and therefore used by current ERP systems.

In REA, transactions are not registered in accounts, but rather as the events that take place. An event in REA is of the form (a_1, a_2, r) meaning that agent a_1 transfers resource r to agent a_2. Hence, when a customer pays an amount x to a company, then it is represented by a single event $(\text{customer}, \text{company}, x)$. Since events are atomic, REA does not have the same redundancy[4] as DEB, and inconsistency is not a possibility: resources always have an origin and a destination. The POETS architecture not only fits with the REA ontology, it is based on it. Events are stored as first-class objects in the event log, and contracts describe the expected future flow of resources.[5] Reports enable computation of derived information that is inherent in DEB, and which may be a legal requirement for auditing. For instance, a sales account—which summarises (pending) sales payments—can be reconstructed from information about initiated sales and payments made by customers. Such a computation will yield the same *derived* information as in DEB, and the principles of DEB consistency will be fulfilled simply by construction.

1.1 Outline and Contributions

The motivation for our work is to assess the POETS architecture in terms of a prototype implementation. During the implementation process we have added features to the architecture that were painfully missing. Moreover, in the process we found that the architecture need not be tailored to the REA ontology—indeed to ERP systems—but the applicability of our generalised architecture to other domains remains future research. Our contributions are as follows:

- We present a generalised and extended POETS architecture (Section 2) that has been fully implemented.
- We present domain-specific languages (DSLs) for data modelling (Section 2.1), report specification (Section 2.4), and contract specification (Section 2.5).
- We demonstrate how to implement a small use case, from scratch, in our implemented system (Section 3). We provide the complete specification of the system, which demonstrates both the conciseness and domain-orientation[6] of our approach. We conclude that the extended architecture is indeed well-suited for implementing ERP systems—although the DSLs and the data model may require additions for larger systems. Most notably, the amount of code needed to implement the system is but a fraction of what would be have to be implemented in a standard ERP system.

[4]In traditional DEB, redundancy is a feature to check for consistency. However, in a computer system such redundancy is superfluous.

[5]Structured contracts are not part of the original REA ontology but instead due to Andersen et al. [1].

[6]Compare the motto: "[...] the formalized requirements *are* the system" [6, page 382].

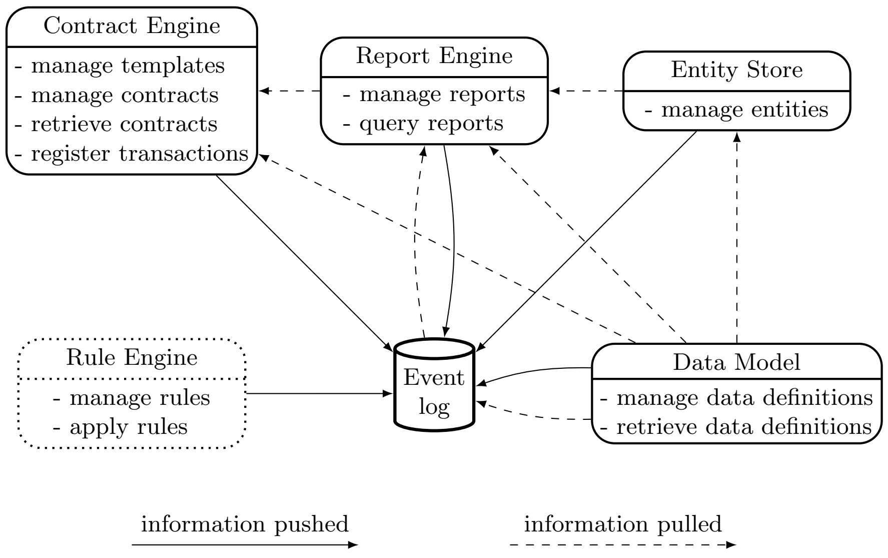

Figure 2: Bird's-eye view of the generalised and extended POETS architecture.

- We describe how we have utilised state-of-the art software development tools in our implementation, especially how the tightly coupled DSLs are implemented (Section 4).

2 Revised POETS Architecture

Our generalised and extended architecture is presented in Figure 2. Compared to the original architecture in Figure 1, the revised architecture sees the addition of three new components: a *data model*, an *entity store*, and a *rule engine*. The rule engine is currently not implemented, and we will therefore not return to this module until Section 5.1.

As in the original POETS architecture, the event log is at the heart of the system. However, in the revised architecture the event log plays an even greater role, as it is the *only* persistent state of the system. This means that the states of all other modules are also persisted in the event log, hence the flow of information from all other modules to the event log in Figure 2. For example, whenever a contract is started or a new report is added to the system, then an event reflecting this operation is persisted in the event log. This, in turn, means that the state of each module can—in principle—be derived from the event log. However, for performance reasons each module—including the event log—maintains its own state in memory.

The addition of a data model constitutes the generalisation of the new architecture over the old architecture. In the data model, data definitions can be added to the system—at run-time—such as data defining customers, resources,

Data Model

Function	**Input**	**Output**
addDataDefs	ontology specification	
getRecordDef	record name	type definition
getSubTypes	record name	list of record names

Figure 3: Data model interface.

or payments. Therefore, the system is not a priori tailored to ERP systems or the REA ontology, but it can be instantiated to that, as we shall see in Section 3.

The entity store is added to the architecture in order to support *entities*—unique "objects" with associated data that may evolve over time. For instance, a concrete customer can suitably be modelled as an entity: Although information attributed to that customer—such as address, or even name—are likely to change over time, it is still the same customer. Moreover, we do not want a copy of the customer data in for instance a sale, but rather a reference to that customer. Hence by modelling customers as entities, we are able to derive, for instance, all transactions in which that customer has participated—even if the customer attributes have changed over time.

We describe each module of the revised architecture in the following subsections. Since we will focus on the revised architecture in the remainder of the text, we will refer to said architecture simply as POETS.

2.1 Data Model

The data model is a core component of the extended architecture, and the interface it provides is summarised in Figure 3. The data model defines the *types* of data that are used throughout the system, and it includes predefined types such as events. Custom types such as invoices can be added to the data model at run-time via *addDataDefs*—for simplicity, we currently only allow addition of types, not updates and deletions. Types define the structure of the data in a running POETS instance manifested as *values*. A value—such as a concrete invoice—is an instance of the data specified by a type. Values are not only communicated between the system and its environment but they are also stored in the event log, which is simply a list of values of a certain type.

2.1.1 Types

Structural data such as payments and invoices are represented as *records*, that is typed finite mappings from field labels to values. Record types define the structure of such records by listing the constituent field labels and their associated types. In order to form a hierarchical ontology of record types, we use a nominal subtyping system [14]. That is, each record type has a unique name, and one type is a subtype of another if and only if stated so explicitly or by transitivity. For instance, a customer can be defined as a subtype of a person, which means that a customer contains all the data of a person, similar to inheritance in object oriented programming.

The choice of nominal types over structural types [14] is justified by the domain: the nominal type associated with a record may have a semantic impact. For instance, the type of customers and premium customers may be structurally equal, but a value of one type is considered different from the other, and clients of the system may for example choose to render them differently. Moreover, the purpose of the rule engine, which we return to in Section 5.1, is to define rules for values of a particular semantic domain, such as invoices. Hence it is wrong to apply these rules to data that happens to have the same structure as invoices. Although we use nominal types to classify data, the DSLs support full record polymorphism [13] in order to minimise code duplication. That is, it is possible for instance to use the same piece of code with customers and premium customers, even if they are not related in the subtyping hierarchy.

The grammar for types is as follows:

$$\begin{array}{rll} T ::= & \mathbf{Bool} \mid \mathbf{Int} \mid \mathbf{Real} \mid \mathbf{String} \mid \mathbf{Timestamp} \mid \mathbf{Duration} & \text{(type constants)} \\ \mid & \mathit{RecordName} & \text{(record type)} \\ \mid & [T] & \text{(list type)} \\ \mid & \langle \mathit{RecordName} \rangle & \text{(entity type)} \end{array}$$

Type constants are standard types Booleans, integers, reals, and strings, and less standard types timestamps (absolute time) and durations (relative time). Record types are named types, and the record typing environment—which we will describe shortly—defines the structure of records. For record types we assume a set $\mathit{RecordName} = \{\mathsf{Customer}, \mathsf{Address}, \mathsf{Invoice}, \dots\}$ of record names ranged over by r. Concrete record types are typeset in sans-serif, and they always begin with a capital letter. Likewise, we assume a set *FieldName* of all field names ranged over by f. Concrete field names are typeset in sans-serif beginning with a lower-case letter.

List types $[\tau]$ represent lists of values, where each element has type τ, and it is the only collection type currently supported. Entity types $\langle r \rangle$ represent entity values that have associated data of type r. For instance, if the record type Customer describes the data of a customer, then a value of type $\langle\mathsf{Customer}\rangle$ is a (unique) customer entity, whose associated Customer data may evolve over time. The type system ensures that a value of an entity type in the system will have associated data of the given type, similar to referential integrity in database systems [3]. We will return to how entities are created and modified in Section 2.3.

A *record typing environment* provides the record types that are available, their subtype relation, and the fields they define.

Definition 2.1. A record typing environment is a tuple $(R, A, F, \rho, \leq)$ consisting of finite sets $R \subseteq \mathit{RecordName}$ and $F \subseteq \mathit{FieldName}$, a set $A \subseteq R$, a mapping $\rho : R \to \mathcal{P}_{\mathrm{fin}}(F \times T)$, and a relation $\leq \subseteq R \times R$, where $\mathcal{P}_{\mathrm{fin}}(X)$ denotes the set of all finite subsets of a set X.

Intuitively, R is the set of defined record types, ρ gives for each defined record type its fields and their types, $\leq$ gives the subtyping relation between record types, and record types in A are considered to be abstract. Abstract record types are not supposed to be instantiated, they are only used to structure the record type hierarchy. The functions *getRecordDef* and *getSubTypes* from Figure 3 provide the means to retrieve the record typing environment from a running system.

Record types can depend on other record types by having them as part of the type of a constituent field:

Definition 2.2. The *immediate dependency relation* of a record typing environment $\mathcal{R} = (R, A, F, \rho, \leq)$, denoted $\rightarrow_{\mathcal{R}}$, is the binary relation on R such that $r_1 \rightarrow_{\mathcal{R}} r_2$ iff there is some $(f, \tau) \in \rho(r_1)$ such that a record name r occurs in τ with $r_2 \leq r$. The *dependency relation* $\rightarrow^+_{\mathcal{R}}$ of $\mathcal{R}$ is the transitive closure of $\rightarrow_{\mathcal{R}}$.

We do not permit all record typing environments. Firstly, we do not allow the subtyping to be cyclic, that is a record type r cannot have a proper subtype which has r as a subtype. Secondly, the definition of field types must be unique and must follow the subtyping, that is a subtype must define at least the fields of its supertypes. Lastly, we do not allow recursive record type definitions, that is a cycle in the dependency relation. The two first restrictions are sanity checks, but the last restriction makes a qualitative difference: the restriction is imposed for simplicity, and moreover we have not encountered practical situations where recursive types were needed.

Definition 2.3. A record typing environment $\mathcal{R} = (R, A, F, \rho, \leq)$ is *well-formed*, whenever the following is satisfied:

- $\leq$ is a partial order, (acyclic inheritance)
- each $\rho(r)$ is the graph of a partial function $F \rightharpoonup T$, (unique typing)
- $r_1 \leq r_2$ implies $\rho(r_1) \supseteq \rho(r_2)$, and (consistent typing)
- $\rightarrow^+_{\mathcal{R}}$ is irreflexive, that is $r_1 \rightarrow^+_{\mathcal{R}} r_2$ implies $r_1 \neq r_2$. (non-recursive)

Well-formedness of a record typing environment combines both conditions for making it easy to reason about them—for instance, transitivity of $\leq$ and inclusion of fields of supertypes—and hard restrictions such as non-recursiveness and unique typing. If a record typing environment fails to be well-formed due to the former only, it can be uniquely closed to a well-formed one:

Definition 2.4. The *closure* of a record typing environment $\mathcal{R} = (R, A, F, \rho, \leq)$ is the record typing environment $\mathrm{Cl}\,(\mathcal{R}) = (R, A, F, \rho', \leq')$ such that $\leq'$ is the transitive, reflexive closure of $\leq$ and ρ' is the consistent closure of ρ with respect to $\leq'$, that is $\rho'(r) = \bigcup_{r \leq' r'} \rho(r')$.

The definition of closure allows us to easily build a well-formed record typing environment from an incomplete specification.

Example 2.5. As an example, we may define a record typing environment $\mathcal{R} = (R, A, F, \rho, \leq)$ for persons and customers as follows:

$$\begin{aligned}
R &= \{\mathsf{Person}, \mathsf{Customer}, \mathsf{Address}\} & \rho(\mathsf{Person}) &= \{(\mathsf{name}, \mathbf{String})\}\\
A &= \{\mathsf{Person}\} & \rho(\mathsf{Customer}) &= \{(\mathsf{address}, \mathsf{Address})\}\\
F &= \{\mathsf{name}, \mathsf{address}, \mathsf{road}, \mathsf{no}\} & \rho(\mathsf{Address}) &= \{(\mathsf{road}, \mathbf{String}), (\mathsf{no}, \mathbf{Int})\},
\end{aligned}$$

with $\mathsf{Customer} \leq \mathsf{Person}$. The only properties that prevent $\mathcal{R}$ from being well-formed are the missing field typing $(\mathsf{name}, \mathbf{String})$ that $\mathsf{Customer}$ should inherit from Person and the missing reflexivity of $\leq$. Hence, the closure $\mathrm{Cl}(\mathcal{R})$ of $\mathcal{R}$ is indeed a well-formed record typing environment.

In order to combine record typing environments we define the union $\mathcal{R}_1 \cup \mathcal{R}_2$ of two record typing environments $\mathcal{R}_i = (R_i, A_i, F_i, \rho_i, \leq_i)$ as the pointwise union:

$$\mathcal{R}_1 \cup \mathcal{R}_2 = (R_1 \cup R_2, A_1 \cup A_2, F_1 \cup F_2, \rho_1 \cup \rho_2, \leq_1 \cup \leq_2),$$

where $(\rho_1 \cup \rho_2)(r) = \rho_1(r) \cup \rho_2(r)$ for all $r \in R_1 \cup R_2$. Note that the union of two well-formed record typing environments need not be well-formed—either due to incompleteness, which can be resolved by forming the closure of the union, or due to inconsistencies respectively cyclic dependencies, which cannot be resolved.

2.1.2 Values

The set of values *Value* supplementing the types from the previous section is defined inductively as the following disjoint union:

$$\mathit{Value} = \mathit{Bool} \uplus \mathit{Int} \uplus \mathit{Real} \uplus \mathit{String} \uplus \mathit{Timestamp} \uplus \mathit{Duration} \uplus \mathit{Record} \uplus \mathit{List} \uplus \mathit{Ent},$$

with

$$\begin{aligned}
\mathit{Bool} &= \{\mathit{true}, \mathit{false}\} & \mathit{String} &= \mathit{Char}^* & \mathit{Record} &= \mathit{RecordName} \times \mathit{Fields}\\
\mathit{Int} &= \mathbb{Z} & \mathit{Timestamp} &= \mathbb{N} & \mathit{Fields} &= \mathit{FieldName} \rightharpoonup_{\text{fin}} \mathit{Value}\\
\mathit{Real} &= \mathbb{R} & \mathit{Duration} &= \mathbb{Z} & \mathit{List} &= \mathit{Value}^*,
\end{aligned}$$

where X^* denotes the set of all finite sequences over a set X; *Char* is a set of characters; *Ent* is an abstract, potentially infinite set of entity values; and $A \rightharpoonup_{\text{fin}} B$ denotes the set of finite partial mappings from a set A to a set B.

Timestamps are modelled using UNIX time[7] and durations are measured in seconds. A record $(r, m) \in \mathit{Record}$ consists of a record name $r \in \mathit{RecordName}$ together with a finite set of named values $m \in \mathit{Fields}$. Entity values $e \in \mathit{Ent}$ are abstract values that only permit equality testing and dereferencing—the latter takes place only in the report engine (Section 2.4), and the type system ensures that dereferencing cannot get stuck, as we shall see in the following subsection.

[7] http://en.wikipedia.org/wiki/Unix_time.

Example 2.6. As an example, a customer record value $c \in \mathit{Record}$ may be as follows:

$$\begin{aligned} c &= (\mathsf{Customer}, m) & m'(\mathsf{road}) &= \texttt{Universitetsparken} \\ m(\mathsf{name}) &= \texttt{John Doe} & m'(\mathsf{no}) &= 1, \\ m(\mathsf{address}) &= (\mathsf{Address}, m') \end{aligned}$$

where $m, m' \in \mathit{Fields}$.

2.1.3 Type Checking

All values are type checked before they enter the system, both in order to check that record values conform with the record typing environment, but also to check that entity values have valid associated data. In particular, events—which are values—are type checked before they are persisted in the event log. In order to type check entities, we assume an *entity typing environment* $\mathcal{E} : \mathit{Ent} \rightharpoonup_{\mathrm{fin}} \mathit{RecordName}$, that is a finite partial mapping from entities to record names. Intuitively, an entity typing environment maps an entity to the record type that it has been declared to have upon creation.

The typing judgement has the form $\mathcal{R}, \mathcal{E} \vdash v : \tau$, where $\mathcal{R}$ is a well-formed record typing environment, $\mathcal{E}$ is an entity typing environment, $v \in \mathit{Value}$ is a value, and $\tau \in T$ is a type. The typing judgment uses the auxiliary subtyping judgement $\mathcal{R} \vdash \tau_1 <: \tau_2$, which is a generalisation of the subtyping relation from Section 2.1.1 to arbitrary types.

The typing rules are given in Figure 4. The typing rules for base types and lists are standard. In order to type check a record, we need to verify that the record contains all and only those fields that the record typing environment prescribes, and that the values have the right type. The typing rule for entities uses the entity typing environment to check that each entity has associated data, and that the data has the required type. The last typing rule enables values to be coerced to a supertype in accordance with the subtyping judgement, which is also given in Figure 4. The rules for the subtyping relation extend the relation from Section 2.1.1 to include subtyping of base types, and contextual rules for lists and entities. We remark that the type system in Figure 4 is declarative: in our implementation, an equivalent algorithmic type system is used.

Example 2.7. Reconsider the record typing environment $\mathcal{R}$ and its closure $\mathrm{Cl}(\mathcal{R})$ from Example 2.5, and the record value c from Example 2.6. Using the typing rules in Figure 4, we can derive the typing judgement $\mathrm{Cl}(\mathcal{R}), \mathcal{E} \vdash c : \mathsf{Customer}$ for any entity typing environment $\mathcal{E}$. Moreover, since $\mathsf{Customer}$ is a subtype of Person we also have that $\mathrm{Cl}(\mathcal{R}), \mathcal{E} \vdash c : \mathsf{Person}$.

In the following, we want to detail how the typing rules guarantee the integrity of entities, which involves reasoning about the evolution of the system over time. To this end, we use $\mathcal{R}_t = (R_t, A_t, F_t, \rho_t, \leq_t)$ and $\mathcal{E}_t$ to indicate the record typing environment and the entity typing environment respectively, at a point in time $t \in \mathit{Timestamp}$. In order to reason about the data associated with

$$\boxed{\mathcal{R},\mathcal{E} \vdash v : \tau} \quad \frac{b \in \mathit{Bool}}{\mathcal{R},\mathcal{E} \vdash b : \mathbf{Bool}} \qquad \frac{n \in \mathit{Int}}{\mathcal{R},\mathcal{E} \vdash n : \mathbf{Int}} \qquad \frac{r \in \mathit{Real}}{\mathcal{R},\mathcal{E} \vdash r : \mathbf{Real}}$$

$$\frac{s \in \mathit{String}}{\mathcal{R},\mathcal{E} \vdash s : \mathbf{String}} \qquad \frac{t \in \mathit{Timestamp}}{\mathcal{R},\mathcal{E} \vdash t : \mathbf{Timestamp}} \qquad \frac{d \in \mathit{Duration}}{\mathcal{R},\mathcal{E} \vdash d : \mathbf{Duration}}$$

$$\frac{(r,m) \in \mathit{Record} \qquad \begin{array}{c}\mathcal{R} = (R, A, F, \rho, \leq) \\ r \in R \setminus A\end{array} \qquad \begin{array}{c}\mathrm{dom}(\rho(r)) = \mathrm{dom}(m) \\ \forall f \in \mathrm{dom}(m)\colon \mathcal{R},\mathcal{E} \vdash m(f) : \rho(r)(f)\end{array}}{\mathcal{R},\mathcal{E} \vdash (r,m) : r}$$

$$\frac{(v_1,\ldots,v_n) \in \mathit{List} \qquad \forall i \in \{1,\ldots,n\}.\mathcal{R},\mathcal{E} \vdash v_i : \tau}{\mathcal{R},\mathcal{E} \vdash (v_1,\ldots,v_n) : [\tau]} \qquad \frac{e \in \mathit{Ent} \qquad \mathcal{E}(e) = r}{\mathcal{R},\mathcal{E} \vdash e : \langle r \rangle}$$

$$\frac{\mathcal{R},\mathcal{E} \vdash v : \tau' \qquad \mathcal{R} \vdash \tau' <: \tau}{\mathcal{R},\mathcal{E} \vdash v : \tau}$$

$$\boxed{\mathcal{R} \vdash \tau_1 <: \tau_2} \quad \frac{}{\mathcal{R} \vdash \tau <: \tau} \qquad \frac{\mathcal{R} \vdash \tau_1 <: \tau_2 \qquad \mathcal{R} \vdash \tau_2 <: \tau_3}{\mathcal{R} \vdash \tau_1 <: \tau_3}$$

$$\frac{}{\mathcal{R} \vdash \mathbf{Int} <: \mathbf{Real}} \qquad \frac{r_1 \leq r_2}{(R, A, F, \rho, \leq) \vdash r_1 <: r_2}$$

$$\frac{\mathcal{R} \vdash \tau_1 <: \tau_2}{\mathcal{R} \vdash [\tau_1] <: [\tau_2]} \qquad \frac{r_1 \leq r_2}{(R, A, F, \rho, \leq) \vdash \langle r_1 \rangle <: \langle r_2 \rangle}$$

Figure 4: Type checking of values $\mathcal{R},\mathcal{E} \vdash v : \tau$ and subtyping $\mathcal{R} \vdash \tau_1 <: \tau_2$.

an entity, we assume for each point in time $t \in \mathit{Timestamp}$ an *entity environment* $\epsilon_t : \mathit{Ent} \rightharpoonup_{\mathrm{fin}} \mathit{Record}$ that maps an entity to its associated data. Entity (typing) environments form the basis of the entity store, which we will describe in detail in Section 2.3.

Given $T \subseteq \mathit{Timestamp}$ and sequences $(\mathcal{R}_t)_{t \in T}$, $(\mathcal{E}_t)_{t \in T}$ and $(\epsilon_t)_{t \in T}$ of record typing environments, entity typing environments, and entity environments respectively, which represent the evolution of the system over time, we require the following invariants to hold for all $t, t' \in \mathit{Timestamp}$, $r, r' \in \mathit{RecordName}$, $e \in \mathit{Ent}$, and $v \in \mathit{Record}$:

$$\text{if } \mathcal{E}_t(e) = r \text{ and } \mathcal{E}_{t'}(e) = r', \text{ then } r = r', \qquad \text{(stable type)}$$

$$\text{if } \mathcal{E}_t(e) \text{ is defined, then so is } \epsilon_t(e), \text{ and} \qquad \text{(well-definedness)}$$

$$\text{if } \epsilon_t(e) = v, \text{ then } \mathcal{E}_t(e) = r \text{ and } \mathcal{R}_{t'}, \mathcal{E}_{t'} \vdash v : r \text{ for some } t' \leq t. \qquad \text{(well-typing)}$$

We refer to the three invariants above collectively as the *entity integrity invariants*. The *stable type* invariant states that each entity can have at most one declared type throughout its lifetime. The *well-definedness* invariant guarantees that every entity that is given a type also has an associated record value.

Finally, the *well-typing* invariant guarantees that the record value associated with an entity *was* well-typed at some earlier point in time t'.

The well-typing invariant is, of course, not strong enough. What we need is that the value v associated with an entity e *remains* well-typed throughout the lifetime of the system. This is, however, dependant on the record typing environment and the entity typing environment, which both may change over time. Therefore, we need to impose restrictions on the possible evolution of the record typing environment, and we need to take into account that entities used in the value v may have been deleted. We return to these issues in Section 2.2 and Section 2.3, and in the latter we will see that the entity integrity invariants are indeed satisfied by the system.

2.1.4 Ontology Language

Section 2.1.1 provides the semantic account of record types, and in order to specify record types, we use a variant of Attempto Controlled English [4] due to Jønsson Thomsen [8], referred to as the *ontology language*. The approach is to define data types in near-English text, in order to minimise the gap between requirements and specification. As an example, the record typing environment from Example 2.5 is specified in the ontology language as follows:

Person is abstract.
Person has a String called name.

Customer is a Person.
Customer has an Address.

Address has a String called road.
Address has an Int called no.

An ontology definition consists of a sequence of sentences as defined by the grammar below (where [·] denotes optionality):

Ontology	::=	*Sentence**	(ontology)
Sentence	::=	*RecordName* **is** [**a** | **an**] *RecordName*.	(supertype declaration)
	|	*RecordName* **is abstract**.	(abstract declaration)
	|	*RecordName* **has** [**a** | **an**] *Type* [**called** *FieldName*].	(field declaration)
Type	::=	**Bool** | **Int** | **Real**	(type constants)
	|	**String** | **Timestamp** | **Duration**	
	|	*RecordName*	(record type)
	|	**list of** *Type*	(list type)
	|	*RecordName* **entity**	(entity type)

The language of types *Type* reflects the definition of types in T and there is an obvious bijection $[\![\cdot]\!] : \mathit{Type} \to T$ with $[\![\textbf{list of } t]\!] = [[\![t]\!]]$, $[\![r \textbf{ entity}]\!] = \langle r \rangle$, and otherwise $[\![t]\!] = t$.

The semantics of the ontology language is given by a straightforward mapping into the domain of record typing environments. Each sentence is translated into a record typing environment. The semantics of a sequence of sentences is simply the closure of the union of each sentence's record typing environment:

$$\begin{aligned}
[\![s_1 \cdots s_n]\!] &= \mathrm{Cl}\left([\![s_1]\!] \cup [\![s_2]\!] \cup \cdots \cup [\![s_n]\!]\right) \\
[\![r_1 \textbf{ is } [\textbf{a} \mid \textbf{an}]\ r_2\textbf{.}]\!] &= \left(\{r_1, r_2\}, \emptyset, \emptyset, \{r_1 \mapsto \emptyset, r_2 \mapsto \emptyset\}, \{(r_1, r_2)\}\right) \\
[\![r \textbf{ is abstract.}]\!] &= \left(\{r\}, \{r\}, \emptyset, \{r \mapsto \emptyset\}, \emptyset\right) \\
[\![r \textbf{ has } [\textbf{a} \mid \textbf{an}]\ t \textbf{ called } f\textbf{.}]\!] &= \left(\{r\}, \emptyset, \{f\}, \{r \mapsto \{(f, [\![t]\!])\}\}, \emptyset\right)
\end{aligned}$$

We omit the case where the optional *FieldName* is not supplied in a field declaration. We treat this form as syntactic sugar for r **has** (**a** | **an**) t **called** f**.** where f is derived from the type t. In this case a default name is used based on the type, simply by changing the first letter to a lower-case. Hence, in the example above the field name of a customer's address is address. Note that the record typing environment need not be well-formed (Definition 2.3), and a subsequent check for well-formedness has to be performed.

Data definitions added to the system via *addDataDefs* are specified in the ontology language. We require, of course, that the result of adding data definitions must yield a well-defined record typing environment. Moreover, we impose further monotonicity constraints which ensure that existing data in the system remain well-typed. We return to these constraints when we discuss the event log in Section 2.2. Type definitions retrieved via *getRecordDef* provide the semantic structure of a record type, that is its immediate supertypes, its fields, and an indication whether the record type is abstract. *getSubTypes* returns a list of immediate subtypes of a given record type, hence *getRecordDef* and *getSubTypes* provide the means for clients of the system to traverse the type hierarchy—both upwards and downwards.

2.1.5 Predefined Ontology

Unlike the original POETS architecture [6], our generalised architecture is not fixed to an enterprise resource planning (ERP) domain. However, we require a set of predefined record types, which are included in Appendix A. That is, the record typing environment $\mathcal{R}_0$ denoted by the ontology in Appendix A is the initial record typing environment in all POETS instances.

The predefined ontology defines five root concepts in the data model, that is record types maximal with respect to the subtype relation $\leq$. Each of these five root concepts Data, Event, Transaction, Report, and Contract are abstract and only Event and Contract define record fields. Custom data definitions added via *addDataDefs* are only permitted as subtypes of Data, Transaction, Report, and Contract. In contrast to that, Event has a predefined and fixed hierarchy.

Data types represent elements in the domain of the system such as customers, items, and resources.

Transaction types represent events that are associated with a contract, such as payments, deliveries, and issuing of invoices.

Report types are result types of report functions, that is the data of reports, such as inventory status, income statement, and list of customers. The Report structure does not define *how* reports are computed, only *what kind* of result is computed. We will return to this discussion in Section 2.4.

Contract types represent the different kinds of contracts, such as sales, purchases, and manufacturing procedures. Similar to Report, the structure does not define what the contract dictates, only what is required to instantiate the contract. The purpose of Contract is hence dual to the purpose of Report: the former determines an input type, and the latter determines an output type. We will return to contracts in Section 2.5.

Event types form a fixed hierarchy and represent events that are logged in the system. Events are conceptually separated into *internal* events and *external* events, which we describe further in the following section.

2.2 Event Log

The event log is the only persistent state of the system, and it describes the complete state of a running POETS instance. The event log is an append-only list of records of the type Event defined in Appendix A. Each event reflects an atomic interaction with the running system. This approach is also applied at the "meta level" of POETS: in order to allow agile evolution of a running POETS instance, changes to the data model, reports, and contracts are reflected in the event log as well.

The monotonic nature of the event log—data is never overwritten or deleted from the system—means that the state of the system can be reconstructed at any previous point in time. In particular, transactions are never deleted, which is a legal requirement for ERP systems. The only component of the architecture that reads directly from the event log is the report engine (compare Figure 2), hence the only way to access data in the log is via a report.

All events are equipped with an internal timestamp (internalTimeStamp), the time at which the event is registered in the system. Therefore, the event log is always monotonically decreasing with respect to internal timestamps, as the newest event is at the head of the list. Conceptually, events are divided into *external* and *internal* events.

External events are events that are associated with a contract, and only the contract engine writes external events to the event log. The event type TransactionEvent models external events, and it consists of three parts: (i) a contract identifier (contractId), (ii) a timestamp (timeStamp), and (iii) a transaction (transaction). The identifier associates the external event with a contract, and the timestamp represents the time at which the external event takes place. Note that the timestamp need not coincide with the internal timestamp. For instance, a payment in a sales contract may be registered in the system the day after it takes place. There is hence no a priori guarantee that external events have decreasing timestamps in the event log—only external events that pertain to the same contract are required to have decreasing timestamps. The

Event	**Description**
AddDataDefs	A set of data definitions is added to the system. The field defs contains the ontology language specification.
CreateEntity	An entity is created. The field data contains the data associated with the entity, the field recordType contains the string representation of the declared type, and the field ent contains the newly created entity value.
UpdateEntity	The data associated with an entity is updated.
DeleteEntity	An entity is deleted.
CreateReport	A report is created. The field code contains the specification of the report, and the fields description and tags are meta data.
UpdateReport	A report is updated.
DeleteReport	A report is deleted.
CreateContractDef	A contract template is created. The field code contains the specification of the contract template, and the fields recordType and description are meta data.
UpdateContractDef	A contract template is updated.
DeleteContractDef	A contract template is deleted.
CreateContract	A contract is instantiated. The field contractId contains the newly created identifier of the contract and the field contract contains the name of the contract template to instantiate, as well as data needed to instantiate the contract template.
UpdateContract	A contract is updated.
ConcludeContract	A contract is concluded.

Figure 5: Internal events.

last component, transaction, represents the actual action that takes place, such as a payment from one person or company to another. The transaction is a record of type Transaction, for which the system has no presumptions.

Internal events reflect changes in the state of the system at a meta level. This is the case for example when a contract is instantiated or when a new record definition is added. Internal events are represented by the remaining subtypes of the Event record type. Figure 5 provides an overview of all non-abstract record types that represent internal events.

A common pattern for internal events is to have three event types to represent creation, update, and deletion of respective components. For instance, when a report is added to the report engine, a CreateReport event is persisted to the log, and when it is updated or deleted, UpdateReport and DeleteReport events are persisted accordingly. This means that previous versions of the report specification can be retrieved, and more generally that the system can be restarted simply by replaying the events that are persisted in the log on an initially empty system. Another benefit to the approach is that the report engine, for instance, does not need to provide built-in functionality to retrieve,

Entity Store

Function	**Input**	**Output**
createEntity	record name, record	entity
updateEntity	entity, record	
deleteEntity	entity	

Figure 6: Entity store interface.

say, the list of all reports added within the last month—such a list can instead be computed as a report itself! We will see how to write such a "meta" report in Section 2.4. Similarly, lists of entities, contract templates, and running contracts can be defined as reports.

Since we allow the data model of the system to evolve over time, we must be careful to ensure that the event log, and thus all data in it, remains well-typed at any point in time. Let $(\mathcal{R}_t)_{t \in T}$, $(\mathcal{E}_t)_{t \in T}$, and $(l_t)_{t \in T}$ be sequences of record typing environments, entity typing environments, and event logs respectively. Since an entity might be deleted over time, and thus is removed from the entity typing environment, the event log may not be well-typed with respect to the current entity typing environment. To this end, we type the event log with respect to the *accumulated entity typing environment* $\widehat{\mathcal{E}}_t = \bigcup_{t' \leq t} \mathcal{E}_{t'}$. That is, $\widehat{\mathcal{E}}_t(e) = r$ iff there is some $t' \leq t$ with $\mathcal{E}_{t'}(e) = r$. The stable type invariant guarantees that $\widehat{\mathcal{E}}_t$ is indeed well-defined.

For changes to the record typing environment, we require the following invariants for any points in time t, t' and the event log l_t at time t:

$$\text{if } t' \geq t \text{ then } \mathcal{R}_{t'} = \mathcal{R}_t \cup \mathcal{R}_\Delta \text{ for some } \mathcal{R}_\Delta\text{, and} \qquad \text{(monotonicity)}$$

$$\mathcal{R}_t, \widehat{\mathcal{E}}_t \vdash l_t : [\mathsf{Event}]\,. \qquad \text{(log typing)}$$

Note that the *log typing* invariant follows from the *monotonicity* invariant and the type checking $\mathcal{R}_t, \mathcal{E}_t \vdash e : \mathsf{Event}$ for each new incoming event, provided that for each record name r occurring in the event log, no additional record fields are added to r, and r is not made an abstract record type. We will refer to the two invariants above collectively as *record typing invariants*. They will become crucial in the following section.

2.3 Entity Store

The entity store provides very simple functionality, namely creation, deletion and updating of entities, respectively. To this end, the entity store maintains the current entity typing environment $\mathcal{E}_t$ as well as the history of entity environments $\epsilon_0, \ldots, \epsilon_t$. The interface of the entity store is summarised in Figure 6.

The creation of a new entity via *createEntity* at time $t+1$ requires a declared type r and an initial record value v, and it is checked that $\mathcal{R}_t, \mathcal{E}_t \vdash v : r$. If the value type checks, a *fresh* entity value $e \notin \bigcup_{t' \leq t} \mathrm{dom}(\epsilon_{t'})$ is created, and the entity environment and the entity typing environment are updated accordingly:

$$\epsilon_{t+1}(x) = \begin{cases} v & \text{if } x = e, \\ \epsilon_t(x) & \text{otherwise,} \end{cases} \qquad \mathcal{E}_{t+1}(x) = \begin{cases} r & \text{if } x = e, \\ \mathcal{E}_t(x) & \text{otherwise.} \end{cases}$$

Moreover, a CreateEntity event is persisted to the event log containing e, r, and v for the relevant fields.

Similarly, if the data associated with an entity e is updated to the value v at time $t+1$, then it is checked that $\mathcal{R}_t, \mathcal{E}_t \vdash v : \mathcal{E}_t(e)$, and the entity store is updated like above. Note that the entity typing environment is unchanged, that is $\mathcal{E}_{t+1} = \mathcal{E}_t$. A corresponding UpdateEntity event is persisted to the event log containing e and v for the relevant fields.

Finally, if an entity e is deleted at time $t+1$, then it is removed from both the entity store and the entity typing environment:

$$\epsilon_{t+1}(x) = \epsilon_t(x) \text{ iff } x \in \operatorname{dom}(\epsilon_t) \setminus \{e\}$$
$$\mathcal{E}_{t+1}(x) = \mathcal{E}_t(x) \text{ iff } x \in \operatorname{dom}(\mathcal{E}_t) \setminus \{e\}\,.$$

A corresponding DeleteEntity event is persisted to the event log containing e for the relevant field.

Note that, by default, $\epsilon_{t+1} = \epsilon_t$ and $\mathcal{E}_{t+1} = \mathcal{E}_t$, unless one of the situations above apply. It is straightforward to show that the *entity integrity invariants* are maintained by the operations described above (the proof follows by induction on the timestamp t). Internally, that is, for the report engine compare Figure 2, the entity store provides a lookup function $\text{lookup}_t : \mathit{Ent} \times [0,t] \rightharpoonup_{\text{fin}} \mathit{Record}$, where $\text{lookup}_t(e, t')$ provides the latest value associated with the entity e at time t', where t is the current time. Note that this includes the case in which e has been deleted at or before time t'. In that case, the value associated with e just before the deletion is returned. Formally, lookup_t is defined in terms of the entity environments as follows:

$$\text{lookup}_t(e, t_1) = v \text{ iff } \exists t_2 \le t_1 \colon \epsilon_{t_2}(e) = v \text{ and } \forall t_2 < t_3 \le t_1 \colon e \notin \operatorname{dom}(\epsilon_{t_3}).$$

In particular, we have that if $e \in \operatorname{dom}(\epsilon_{t_1})$ then $\text{lookup}_t(e, t_1) = \epsilon_{t_1}(e)$.

From this definition and the invariants of the system, we obtain the following property:

Corollary 2.8. *Let $(\mathcal{R}_t)_{t\in T}$, $(\mathcal{E}_t)_{t\in T}$, and $(\epsilon_t)_{t\in T}$ be sequences of record typing environments, entity typing environments, and entity environments respectively, satisfying the entity integrity invariants and the record typing invariants. Then the following holds for all timestamps $t \le t_1 \le t_2$ and entities $e \in \mathit{Ent}$:*

If $\mathcal{R}_t, \widehat{\mathcal{E}}_t \vdash e : \langle r \rangle$ then $\text{lookup}_{t_2}(e, t_1) = v$ for some v and $\mathcal{R}_{t_2}, \widehat{\mathcal{E}}_{t_2} \vdash v : r$.

Proof. Assume that $\mathcal{R}_t, \widehat{\mathcal{E}}_t \vdash e : \langle r \rangle$. Then it follows from the typing rule for entity values and the subtyping rules that $\widehat{\mathcal{E}}_t(e) = r'$ for some r' with $r' \le_t r$. That is, there is some $t' \le t$ with $\mathcal{E}_{t'}(e) = r'$. Hence, from the well-definedness

Report Engine		
Function	**Input**	**Output**
addReport	name, type, description, tags, report definition	
updateReport	name, type, description, tags, report definition	
deleteReport	name	
queryReport	name, list of values	value

Figure 7: Report engine interface.

invariant it follows that $\epsilon_{t'}(e)$ is defined. Since $t' \leq t \leq t_1$, we can thus conclude that $\text{lookup}_{t_2}(e, t_1) = (r'', m)$, for some record value (r'', m).

According to the definition of lookup_{t_2}, we then have some $t_3 \leq t_1$ with $\epsilon_{t_3}(e) = (r'', m)$. Applying the well-typing invariant, we obtain some $t_4 \leq t_3$ with $\mathcal{R}_{t_4}, \mathcal{E}_{t_4} \vdash (r'', m) : \mathcal{E}_{t_3}(e)$. Since, by the stable type invariant, $\mathcal{E}_{t_3}(e) = \mathcal{E}_{t'} = r'$, we then have $\mathcal{R}_{t_4}, \mathcal{E}_{t_4} \vdash (r'', m) : r'$. Moreover, according to the typing rules, this can only be the case if $r'' \leq_{t_4} r'$.

Due to the monotonicity invariant, we know that $\mathcal{R}_{t_2} = \mathcal{R}_{t_4} \cup \mathcal{R}_\Delta$ for some $\mathcal{R}_\Delta$. In particular, this means that $r'' \leq_{t_4} r'$ implies that $r'' \leq_{t_2} r'$. Similarly, $r' \leq_t r$ implies that $r' \leq_{t_2} r$. Hence, by transitivity of $\leq_{t_2}$, we have that $r'' \leq_{t_2} r$.

According to the implementation of the entity store, we know that $\epsilon_{t_3}(e) = (r'', m)$ implies that (r'', m) occurs in the event log (as part of an event of type $\mathsf{CreateEntity}$ or $\mathsf{UpdateEntity}$) at least from t_3 onwards, in particular in the event log l_{t_2} at t_2. Since, by the log typing invariant, the event log l_{t_2} is well-typed as $\mathcal{R}_{t_2}, \widehat{\mathcal{E}_{t_2}} \vdash l_{t_2} : [\mathsf{Event}]$, we know that $\mathcal{R}_{t_2}, \widehat{\mathcal{E}_{t_2}} \vdash (r'', m) : r''$. From the subtype relation $r'' \leq_{t_2} r$ we can thus conclude $\mathcal{R}_{t_2}, \widehat{\mathcal{E}_{t_2}} \vdash (r'', m) : r$. □

The corollary above describes the fundamental safety property with respect to entity values: if an entity value previously entered the system, and hence type checked, then all future dereferencing will not get stuck, and the obtained value will be well-typed with respect to the accumulated entity typing environment.

2.4 Report Engine

The purpose of the report engine is to provide a structured view of the database that is constituted by the system's event log. This structured view of the data in the event log comes in the form of a *report*, which provides a collection of condensed structured information compiled from the event log. Conceptually, the data provided by a report is compiled from the event log by a function of type $[\mathsf{Event}] \to \mathsf{Report}$, a *report function*. The *report language* provides a means to specify such a report function in a declarative manner. The interface of the report engine is summarised in Figure 7.

2.4.1 The Report Language

In this section, we provide an overview over the report language. For a detailed description of the language including the full static and dynamic semantics consult Appendix B.

The report language is—much like the query fragment of *SQL*—a functional language *without side effects*. It only provides operations to non-destructively manipulate and combine values. Since the system's storage is based on a shallow event log, the report language must provide operations to relate, filter, join, and aggregate pieces of information. Moreover, as the data stored in the event log is inherently heterogeneous—containing data of different kinds—the report language offers a comprehensive type system that allows us to safely operate in this setting.

Example 2.9. Consider the following simple report function that lists all reports available in the system:

```
reports : [PutReport]
reports = nubProj (λx → x.name) [pr |
  cr : CreateReport ← events,
  pr : PutReport = first cr [ur | ur : ReportEvent ← events,
                                   ur.name ≡ cr.name]]
```

The report function above uses the two functions *nubProj* and *first*, which are defined in the standard library of the report language. The function *nubProj* of type $(\mathsf{Eq}\ b) \Rightarrow (a \to b) \to [a] \to [a]$ removes duplicates in the given list according to the equality on the result of the provided projection function. In the example above, reports with the same name are considered duplicates. The function $\mathit{first} : a \to [a] \to a$ returns the first element of the given list or the default value provided as first argument if the list is empty.

Every report function implicitly has as its first argument the event log of type [Event]—a list of events—bound to the name **events**. The syntax—and to large parts also the semantics—is based on Haskell [9]. The central data structure is that of lists. In order to formulate operations on lists concisely, we use list comprehensions [16] as seen in Example 2.9. A list comprehension of the form $[\ e \mid c\ ]$ denotes a list containing elements of the form e generated by c, where c is a sequence of *generators* and *filters*.

As we have mentioned, access to type information and its propagation to subsequent computations is essential due to the fact that the event log is a list of heterogeneously typed elements—events of different kinds. The generator cr : CreateReport ← **events** iterates through elements of the list **events**, binding each element to the variable cr. The typing cr : CreateReport restricts this iteration to elements of type CreateReport, a subtype of Event. This type information is propagated through the subsequent generators and filters of the list comprehension. In the filter $\mathit{ur.name} \equiv \mathit{cr.name}$, we use the fact that elements of type ReportEvent have a field name of type **String**. When binding the first element of the result of the nested list comprehension to the variable pr it is also checked whether this element is in fact of type PutReport. Thus we ignore reports that are marked as deleted via a DeleteReport event.

The report language is based on the simply typed lambda calculus extended with polymorphic (non-recursive) let expressions as well as type case expressions. The core language is given by the following grammar:

$$e ::= x \mid c \mid \lambda x.e \mid e_1\ e_2 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid \textbf{type}\ x = e\ \textbf{of}\ \{r \to e_1; \_ \to e_2\}\,,$$

where x ranges over variables, and c over constants which include integers, Booleans, tuples and list constructors as well as operations on them like $+$, *if-then-else* etc. In particular, we assume a fold operation **fold** of type $(\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta$. This is the only operation of the report language that permits recursive computations on lists. List comprehensions are mere syntactic sugar and can be reduced to **fold** and let expressions as for example in Haskell [9].

The extended list comprehensions of the report language that allow filtering according to run-time type information depend on type case expressions of the form $\textbf{type}\ x = e\ \textbf{of}\ \{r \to e_1; \_ \to e_2\}$. In such a type case expression, an expression e of some record type r_e gets evaluated to record value v which is then bound to a variable x. The record type r that the record value v is matched against can be any subtype of r_e. Further evaluation of the type case expression depends on the type r_v of the record value v. This type can be any subtype of r_e. If $r_v \le r$ then the evaluation proceeds with e_1, otherwise with e_2. Binding e to a variable x allows us to use the stricter type r in the expression e_1.

Another important component of the report language consists of the dereferencing operators ! and @, which give access to the lookup operator provided by the entity store. Given an expression e of an entity type $\langle r\rangle$, both dereferencing operators provide a value v of type r. That is, both ! and @ are unary operators of type $\langle r\rangle \to r$ for any record type r. In the case of the operator !, the resulting record value v is the latest value associated with the entity to which e evaluates. More concretely, given an entity value v, the expression $v!$ evaluates to the record value $\mathrm{lookup}_t(v, t)$, where t is the current timestamp.

On the other hand, the *contextual* dereference operator @ provides as the result the value associated with the entity at the moment the entity was used in the event log (based on the internalTimeStamp field). This is the case when the entity is extracted from some event from the event log. Otherwise, the entity value stems from an actual argument to the report function. In the latter case @ behaves like the ordinary dereference operator !. In concrete terms, every entity value v that enters the event log is annotated with the timestamp of the event it occurs in. That is, each entity value embedded in an event e in the event log, occurs in an annotated form (v, s), where s is the value of e's internalTimeStamp field. Given such an annotated entity value (v, s), the expression $(v,s)@$ evaluates to $\mathrm{lookup}_t(v, s)$ and given a bare entity value v the expression $v@$ evaluates to $\mathrm{lookup}_t(v, t)$.

Note that in each case for either of the two dereference operators, Corollary 2.8 guarantees that the lookup operation yields a record value of the right type. That is, both $! : \langle r \rangle \to r$ and $@ : \langle r \rangle \to r$ are total functions that never get stuck.

Example 2.10. The entity store and the contextual dereferencing operator provide a solution to a recurring problem in ERP systems, namely how to maintain historical data for auditing. For example, when an invoice is issued in a sale, then a copy of the customer information *at the time* of the invoice is needed for auditing. Traditional ERP systems solve the problem by explicit copying of data, since referenced data might otherwise get destructively updated.

Since data is never deleted in a POETS system, we can solve the problem without copying. Consider the following definition of transactions that represent issuing of invoices, and invoices respectively (we assume that the record types Customer and OrderLine are already defined):

IssueInvoice is a Transaction.
IssueInvoice has a Customer entity.
IssueInvoice has a list of OrderLine.

Invoice is Data.
Invoice has a Customer.
Invoice has a list of OrderLine.

Rather than containing a Customer record, an IssueInvoice transaction contains a Customer entity, which eliminates copying of data. From an IssueInvoice transaction we can instead *derive* the invoice data by the following report function:

$$\begin{aligned}&\mathit{invoices} : [\mathsf{Invoice}]\\&\mathit{invoices} = [\mathsf{Invoice}\{\mathit{customer} = \mathit{ii.customer@},\ \mathit{orderLines} = \mathit{ii.orderLines}\} \mid\\&\quad \mathit{tr} : \mathsf{TransactionEvent} \leftarrow \mathbf{events},\\&\quad \mathit{ii} : \mathsf{IssueInvoice} = \mathit{tr.transaction}]\end{aligned}$$

Note how the @ operator is used to dereference the customer data: since the *ii.customer* value originates from an event in the event log, the contextual dereferencing will produce data associated with the customer at the time when the invoice was issued, as required.

2.4.2 Incrementalisation

While the type system is important in order to avoid obvious specification errors, it is also important to ensure a fast execution of the thus obtained functional specifications. This is, of course, a general issue for querying systems. In our system it is, however, of even greater importance since shifting the structure of the data—from the data store to the domain of queries—means that queries operate on the complete data set of the database. In principle, the data of each report has to be recomputed after each transaction by applying the corresponding report function to the updated event log. In other words, if treated naïvely, the conceptual simplification provided by the flat event log has to be paid via more expensive computations.

This issue can be addressed by transforming a given report function f into an incremental function f' that updates the report data computed previously according to the changes that have occurred since the report data was computed before. That is, given an event log l and an update to it $l \oplus e$, we require that

$f(l \oplus e) = f'(f(l), e)$. The new report data $f(l \oplus e)$ is obtained by updating the previous report data $f(l)$ according to the changes e. In the case of the event log, we have a list structure. Changes only occur *monotonically*, by adding new elements to it: given an event log l and a new event e, the new event log is $e \# l$, where $\#$ is the list constructor of type $\alpha \to [\alpha] \to [\alpha]$.

Here it is crucial that we have restricted the report language such that operations on lists are limited to the higher-order function **fold**. The fundamental idea of incrementalising report functions is based on the following equation satisfied by **fold**:

$$\textbf{fold}\ f\ e\ (x \# xs) = f\ x\ (\textbf{fold}\ f\ e\ (xs))$$

Based on this idea, we are able to make the computation of most report functions independent of the size of the event log but only dependent of the changes to the event log and the previous result of the report function [12]. Unfortunately, if we consider for example list comprehensions containing more than one generator, we have functions with nested folds. In order to properly incrementalise such functions, we need to move from list structures to multisets. This is, however, only rarely a practical restriction since most aggregation functions are based on commutative binary operations and are thus oblivious to ordering.

2.4.3 Lifecycle of Reports

Like entities, the set of reports registered in a running POETS instance—and thus available for querying—can be changed via the external interface to the report engine. To this end, the report engine interface provides the operations *addReport*, *updateReport*, and *deleteReport*. The former two take a *report specification* that contains the name of the report, the definition of the report function that generates the report data and the type of the report function. Optionally, it may also contain further meta information in the form of a description text and a list of tags.

Example 2.11. Reconsider the function defined in Example 2.9 that lists all active reports with all their meta data. The following report specification uses the report function from Example 2.9 in order to define a report function that lists the names of all active report:

```
name: ReportNames
description: A list of names of all registered reports.
tags: internal, report

reports : [PutReport]
reports = nubProj (λx → x.name) [pr |
  cr : CreateReport ← events,
  pr : PutReport = first cr [ur | ur : ReportEvent ← events,
                                  ur.name ≡ cr.name]]

report : [String]
```

Contract Engine

Function	**Input**	**Output**
createTemplate	name, type, description, specification	
updateTemplate	name, type, description, specification	
deleteTemplate	name	
createContract	meta data	contract ID
updateContract	contract ID, meta data	
concludeContract	contract ID	
getContract	contract ID	contract state
registerTransaction	contract ID, timestamp, transaction	

Figure 8: Contract engine interface.

report = [*r.name* | *r* ← *reports*]

In the header of the report specification, the name and optionally also a description text as well as a list of tags is provided as meta data to the actual report function specification. Every report specification must define a top-level function called **report**, which provides the report function that derives the report data from the event log. In the example above, this function takes no (additional) arguments and returns a list of strings—the names of active reports.

Calls to *addReport* and *updateReport* are both reflected by a corresponding event of type CreateReport and UpdateReport respectively. Both events are subtypes of PutReport and contain the meta information as well as the original specification text of the concerning report. When a report is no longer needed, it can be removed from the report engine by a corresponding *deleteReport* operation. Note that the change and removal of reports only affect the state of the POETS system from the given point in time. Transactions that occurred prior to a change or deletion of a report are not affected. This is important for the system's ability to fully recover after a crash by replaying the events from the event log.

The remaining operation provided by the report engine—*queryReport*—constitutes the core functionality of the reporting system. Given a name of a registered report and a list of arguments, this operation supplies the given arguments to the corresponding report function and returns the result. For example, the *ReportNames* report specified in Example 2.11 does not require any arguments—its type is [**String**]—and returns the names of registered reports.

2.5 Contract Engine

The role of the contract engine is to determine which transactions—that is external events, compare Section 2.2—are expected by the system. Transactions model events that take place according to an *agreement*, for instance a delivery of goods in a sale, a payment in a lease agreement, or a movement of items from one inventory to another in a production plan. Such agreements are referred to as *contracts*, although they need not be legally binding contracts. The purpose of a contract is to provide a detailed description of *what* is expected, by *whom*, and *when*. A sales contract, for example, may stipulate that first the company sends an invoice, then the customer pays within a certain deadline, and finally the company delivers goods within another deadline.

The interface of the contract engine is summarised in Figure 8.

2.5.1 Contract Templates

In order to specify contracts such as the aforementioned sales contract, we use an extended variant of the contract specification language (CSL) of Hvitved et al. [7], which we will refer to as the POETS contract specification language (PCSL) in the following. For reusability, contracts are always specified as *contract templates* rather than as concrete contracts. A contract template consists of four parts: (i) a template name, (ii) a template type, which is a subtype of the Contract record type, (iii) a textual description, and (iv) a PCSL specification. We describe PCSL in Section 2.5.3.

The template name is a unique identifier, and the template type determines the parameters that are available in the contract template.

Example 2.12. We may define the following type for sales contracts in the ontology language (assuming that the record types Customer, Company, and Goods have been defined):

Sale is a Contract.
Sale has a Customer entity.
Sale has a Company entity.
Sale has a list of Goods.
Sale has an Int called amount.

With this definition, contract templates of type Sale are parametrised over the fields customer, company, goods, and amount of types $\langle$Customer$\rangle$, $\langle$Company$\rangle$, [Goods], and **Int**, respectively.

The contract engine provides an interface to add contract templates (*createTemplate*), update contract templates (*updateTemplate*), and remove contract templates (*deleteTemplate*) from the system at run-time. The structure of contract templates is reflected in the external event types CreateContractDef, UpdateContractDef, and DeleteContractDef, compare Section 2.2. A list of (non-deleted) contract templates can hence be computed by a report, similar to the list of (non-deleted) reports from Example 2.11.

2.5.2 Contract Instances

A contract template is instantiated via *createContract* by supplying a record value v of a subtype of Contract. Besides custom fields, which depend on the type at hand, such a record always contains the fields templateName and startDate inherited from the Contract record type, compare Appendix A. The field templateName contains the name of the template to instantiate, and the field startDate determines the start date of the contract. The fields of v are substituted into the contract template in order to obtain a *contract instance*, and the type of v must therefore match the template type. For instance, if v has type Sale then the field templateName must contain the name of a contract template that has type Sale. We refer to the record v as *contract meta data*.

When a contract c is instantiated by supplying contract meta data v, a *fresh* contract identifier i is created, and a CreateContract event is persisted in the event log with with $\mathsf{contract} = v$ and $\mathsf{contractId} = i$. Hereafter, transactions t can be registered with the contract via *registerTransaction*, which will update the contract to a *residual contract* c', written $c \xrightarrow{t} c'$, and a TransactionEvent with $\mathsf{transaction} = t$ and $\mathsf{contractId} = i$ is written to the event log. The state of the contract can be acquired from the contract engine at any given point in time via *getContract*, which enables run-time analyses of contracts, for instance in order to generate a list of expected transactions.

Registration of a transaction $c \xrightarrow{t} c'$ is only permitted if the transaction is expected in the current state c. That is, there need not be a residual state for all transactions. After zero or more successful transactions, $c \xrightarrow{t_1} c_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} c_n$, the contract may be concluded via *concludeContract*, provided that the residual contract c_n does not contain any outstanding obligations. This results in a ConcludeContract event to be persisted in the event log.

The lifecycle described above does not take into account that contracts may have to be updated at run-time, for example if it is agreed to extend the payment deadline in a sales contract. To this end, running contracts are allowed to be updated, simply by supplying new contract meta data (*updateContract*). The difference in the new meta data compared to the old meta data may not only be a change of, say, items to be sold, but it may also be a change in the field templateName. The latter makes it is possible to replace the old contract by a qualitatively different contract, since the new contract template may describe a different workflow. There is, however, an important restriction: a contract can only be updated if any previous transactions registered with the contract also conform with the new contract. That is, if the contract has evolved like $c \xrightarrow{t_1} c_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} c_n$, and an update to a new contract c' is requested, then only if $c' \xrightarrow{t_1} c'_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} c'_n$, for some $c'_1, \ldots, c'_n$, is the update permitted. A successful update results in an UpdateContract event to be written to the event log with the new meta data.

Note that, for simplicity, we only allow the updates described above. Another possibility is to allow updates where the current state of the contract c is replaced directly by a new state c'. Although we can achieve this effect via a suitably defined contract template and the *updateContract* function above, a direct update is preferable.

As for contract templates, a list of (non-concluded) contract instances can be computed by a report that inspects CreateContract, UpdateContract, and ConcludeContract events respectively.

2.5.3 The Contract Language

The fourth component of contract templates—the PCSL specification—is the actual normative content of contract templates. The core grammar for PCSL is presented in Figure 9. PCSL extends CSL mainly at the level of expressions E, by adding support for the value types in POETS, as well as lambda abstractions and function applications. At the level of clauses C, PCSL is similar to CSL, albeit with a slightly altered syntax.

The semantics of PCSL is a straightforward extension of that of CSL [7], although we use a partial small-step semantics rather than CSL's total small-step semantics. That is, there need not be a residue for all clauses and transactions, as described in Section 2.5.2. This is simply in order to prevent "unexpected" events from entering the system, for instance we only allow a payment to be entered into the system if a running contract expects that payment.

The type system for clauses is identical with CSL. Typing of expressions is, however, more challenging since we have introduced (record) polymorphism as well as subtyping. We will not present the extended semantics nor the extended typing rules, but only remark that the typing serves the same purpose as in CSL: evaluation of expressions does not get stuck and always terminates, and contracts have unique blame assignment.

Example 2.13. We demonstrate PCSL by means of an example, presented in Figure 10. The contract template is of the type Sale from Example 2.12, which means that the fields goods, amount, company, and customer are available in the body of the contract template, that is the right-hand side of the **contract** keyword. Hence, concrete values are substituted from the contract meta data when the template is instantiated, as described in Section 2.5.2.

The example uses standard syntactic sugar at the level of expressions, for instance $\neg e$ means **if** e **then** *false* **else** *true* and $e_1 \vee e_2$ means $\neg(\neg e_1 \wedge \neg e_2)$. Moreover, we omit the **after** part of a deadline if it is 0, we write **immediately** for **within** 0, we omit the **remaining** part if it is not used, and we write **fun** $f\, x_1 \cdots x_n = e$ for **val** $f = \lambda x_1 \rightarrow \cdots \lambda x_n \rightarrow e$.

The template implements a simple workflow: first the company issues an invoice, then the customer pays within 14 days, and simultaneously the company delivers goods within a week. Delivery of goods is allowed to take place in multiple deliveries, which is coded as the recursive clause template *delivery*. Note how the variable r is bound to the remainder of the deadline: All deadlines in a **then** branch are relative to the time of the guarding event, hence the relative deadline for delivering the remaining goods is whatever remains of the original one week deadline. Note also that the initial reference time of a contract instance is determined by the field startDate in the contract meta data, compare Appendix A. Hence if the contract above is instantiated with start date $t \in$ *Timestamp*, then the invoice is supposed to be issued at time t.

Finally, we remark that obligation clauses are binders. That is, for instance the variable g is bound to value of the field goods of the IssueInvoice transaction when it takes place, and the scope of g is the **where** clause and the continuation clause following the **then** keyword.

$$
\begin{array}{lll r}
\mathit{Tmp} & ::= & \mathbf{name} : \mathit{ContractName} & \text{(contract template)} \\
 & & \mathbf{type} : \mathit{RecordName} & \\
 & & \mathbf{description} : \mathit{String} & \\
 & & \mathit{Def} \ldots \mathit{Def}\ \mathbf{contract} = C & \\
\mathit{Def} & ::= & \mathbf{val}\ \mathit{Var} = E & \text{(value definition)} \\
 & \mid & \mathbf{clause}\ \mathit{ClauseName}(\mathit{Var} : T, \ldots, \mathit{Var} : T) & \text{(clause template)} \\
 & & \qquad\qquad \langle \mathit{Var} : T, \ldots, \mathit{Var} : T\rangle = C & \\
C & ::= & \mathbf{fulfilment} & \text{(no obligations)} \\
 & \mid & \langle E\rangle\ \mathit{RecordName}(F, \ldots, F) & \text{(obligation)} \\
 & & \ \mathbf{where}\ E\ \mathbf{due}\ D\ \mathbf{remaining}\ \mathit{Var}\ \mathbf{then}\ C & \\
 & \mid & \mathbf{when}\ \mathit{RecordName}(F, \ldots, F) & \text{(external choice)} \\
 & & \ \mathbf{where}\ E\ \mathbf{due}\ D\ \mathbf{remaining}\ \mathit{Var}\ \mathbf{then}\ C\ \mathbf{else}\ C & \\
 & \mid & \mathbf{if}\ E\ \mathbf{then}\ C\ \mathbf{else}\ C & \text{(internal choice)} \\
 & \mid & C\ \mathbf{and}\ C & \text{(conjunction)} \\
 & \mid & C\ \mathbf{or}\ C & \text{(disjunction)} \\
 & \mid & \mathit{ClauseName}(E, \ldots, E)\langle E, \ldots, E\rangle & \text{(instantiation)} \\
F & ::= & \mathit{FieldName}\ \mathit{Var} & \text{(field binder)} \\
R & ::= & \mathit{RecordName}\ \mathit{Var} & \text{(record binder)} \\
T & ::= & \mathit{TypeVar} & \text{(type variable)} \\
 & \mid & () & \text{(unit type)} \\
 & \mid & \mathbf{Bool} \mid \mathbf{Int} \mid \mathbf{Real} \mid \mathbf{String} & \text{(type constants)} \\
 & \mid & \mathbf{Timestamp} \mid \mathbf{Duration} & \\
 & \mid & \mathit{RecordName} & \text{(record type)} \\
 & \mid & [T] & \text{(list type)} \\
 & \mid & \langle T\rangle & \text{(entity type)} \\
 & \mid & T \to T & \text{(function type)} \\
E & ::= & \mathit{Var} & \text{(variable)} \\
 & \mid & \mathit{BaseValue} & \text{(base value)} \\
 & \mid & \mathit{RecordName}\{\mathit{FieldName} = E, \ldots, \mathit{FieldName} = E\} & \text{(record expression)} \\
 & \mid & [E, \ldots, E] & \text{(list expression)} \\
 & \mid & \lambda\, \mathit{Var} \to E & \text{(function abstraction)} \\
 & \mid & E\ E & \text{(function application)} \\
 & \mid & E \oplus E & \text{(binary expression)} \\
 & \mid & E.\mathit{FieldName} & \text{(field projection)} \\
 & \mid & E\{\mathit{FieldName} = E\} & \text{(field update)} \\
 & \mid & \mathbf{if}\ E\ \mathbf{then}\ E\ \mathbf{else}\ E & \text{(conditional)} \\
 & \mid & \mathbf{case}\ E\ \mathbf{of}\ R \to E \mid \cdots \mid R \to E & \text{(record type casing)} \\
D & ::= & \mathbf{after}\ E\ \mathbf{within}\ E & \text{(deadline expression)} \\
\oplus & ::= & \times \mid / \mid + \mid \langle\times\rangle \mid \langle+\rangle \mid \# \mid \equiv \mid \leq \mid \wedge & \text{(binary operators)}
\end{array}
$$

Figure 9: Grammar for the core contract language PCSL. *ContractName* is the set of all contract template names, *ClauseName* is the set of all clause template names ranged over by k, *Var* is the set of all variable names ranged over by x, *TypeVar* is the set of all type variable names ranged over by α, and $\mathit{BaseValue} = \mathit{Bool} \uplus \mathit{Int} \uplus \mathit{Real} \uplus \mathit{String} \uplus \mathit{Timestamp} \uplus \mathit{Duration} \uplus \mathit{Ent}$.

```
name: salesContract
type: Sale
description: "A simple sales contract between a company and a customer"

fun elem x = foldr (λy b → x ≡ y ∨ b) false
fun filter f = foldr (λx b → if f x then x # b else b) []
fun subset l1 l2 = all (λx → elem x l2) l1
fun diff l1 l2 = filter (λx → ¬ (elem x l2)) l1

clause sale(goods : [Goods], amount : Int)⟨comp : ⟨Company⟩, cust : ⟨Customer⟩⟩ =
⟨comp⟩ IssueInvoice(goods g, amount a)
  where g ≡ goods ∧ a ≡ amount due immediately
 then
⟨cust⟩ Payment(amount a)
  where a ≡ amount due within 14D
 and
 delivery(goods, 1W)⟨comp⟩

clause delivery(goods : [Goods], deadline : Duration)⟨comp : ⟨company⟩⟩ =
 if goods ≡ [] then
  fulfilment
 else
  ⟨comp⟩ Delivery(goods g)
    where g ≢ [] ∧ subset g goods due within deadline remaining r
  then
  delivery(diff goods g, r)⟨comp⟩

contract = sale(goods, amount)⟨company, customer⟩
```

Figure 10: PCSL sales contract template of type Sale.

Built-in symbols PCSL has a small set of built-in symbols, from which other standard functions can be derived:

$foldl : (a \to b \to a) \to a \to [b] \to a$
$foldr : (a \to b \to b) \to b \to [a] \to b$
$ceil : \mathsf{Real} \to \mathbf{Int}$
$reports : \mathsf{Reports}$

The list includes fold operations in order to iterate over lists, since explicit recursion is not permitted, and a special constant *reports* of type Reports. The record type Reports is internally derived from the active reports in the report engine, and it is used only in the contract engine in order to enable querying of reports from within contracts. The record type contains one field per report. For instance, if the report engine contains a single report *Inventory* of type Inventory, then the typing of Report is (using the same notation as in Section 2.1.1):

$$\rho(\mathsf{Reports}) = \{(\mathsf{inventory}, () \to \mathsf{Inventory})\},$$

and the expression *reports.inventory* () invokes the report.

3 Use Case: μERP

In this section we describe a use case instantiation of POETS, which we refer to as μERP. With μERP we model a simple ERP system for a small bicycle shop. Naturally, we do not intend to model all features of a full-blown ERP system, but rather we demonstrate a limited set of core ERP features. In our use case, the shop purchases bicycles from a bicycle vendor, and sells those bicycles to customers. We want to make sure that the bicycle shop only sells bicycles in stock, and we want to model a repair guarantee, which entitles customers to have their bikes repaired free of charge up until three months after purchase.

Following Henglein et al. [6], we also provide core financial reports, namely the income statement, the balance sheet, the cash flow statement, the list of open (not yet paid) invoices, and the value-added tax (VAT) report. These reports are typical, minimal legal requirements for running a business. We provide some example code in this section, and the complete specification is included in Appendix C. As we have seen in Section 2, instantiating POETS amounts to defining a data model, a set of reports, and a set of contract templates. We describe each of these components in the following subsections.

3.1 Data Model

The data model of μERP is tailored to the ERP domain in accordance with the REA ontology [10]. Therefore, the main components of the data model are resources, transactions (that is, events associated with contracts), and agents. The complete data model is provided in Appendix C.1.

Agents are modelled as an abstract type Agent. An agent is either a Customer, a Vendor, or a special Me agent. Customers and Vendors are equipped with a name and an address. The Me type is used to represent the bicycle company itself. In a more elaborate example, the Me type will have subtypes such as Inventory or SalesPerson to represent subdivisions of, or individuals in, the company. The agent model is summarised below:

```
Agent is Data.                            Me is an Agent.

Customer is an Agent.                     Vendor is an Agent.
Customer has a String called name.        Vendor has a String called name.
Customer has an Address.                  Vendor has an Address.
```

Resources are—like agents—Data. In our modelling of resources, we make a distinction between *resource types* and *resources*. A resource type represents a kind of resource, and resource types are divided into currencies (Currency) and item types (ItemType). Since we are modelling a bicycle shop, the only item type (for now) is bicycles (Bicycle). A resource is an instance of a resource type, and—similar to resource types—resources are divided into money (Money) and items (Item). Our modelling of items assumes an implicit unit of measure, that is we do not explicitly model units of measure such as pieces, boxes, pallets, etc. Our resource model is summarised below:

ResourceType is Data.
ResourceType is abstract.

Currency is a ResourceType.
Currency is abstract.

DKK is a Currency.
EUR is a Currency.

ItemType is a ResourceType.
ItemType is abstract.

Bicycle is an ItemType.

Bicycle has a String called model.

Resource is Data.
Resource is abstract.

Money is a Resource.
Money has a Currency.
Money has a Real called amount.

Item is a Resource.
Item has an ItemType.
Item has a Real called quantity.

Transactions (events in the REA terminology) are, not surprisingly, subtypes of the built-in Transaction type. The only transactions we consider in our use case are bilateral transactions (BiTransaction), that is transactions that have a sender and a receiver. Both the sender and the receiver are agent entities, that is a bilateral transaction contains references to two agents rather than copies of agent data. For our use case we model payments (Payment), deliveries (Delivery), issuing of invoices (IssueInvoice), requests for repair of a set of items (RequestRepair), and repair of a set of items (Repair). Issuing of invoices contain the relevant information for modelling of VAT, encapsulated in the OrderLine type. We include some of these definitions below:

BiTransaction is a Transaction.
BiTransaction is abstract.
BiTransaction has an Agent entity called sender.
BiTransaction has an Agent entity called receiver.

Transfer is a BiTransaction.
Transfer is abstract.

Payment is a Transfer.

Payment is abstract.
Payment has Money.

CashPayment is a Payment.
CreditCardPayment is a Payment.
BankTransfer is a Payment.

IssueInvoice is a BiTransaction.
IssueInvoice has a list of OrderLine called orderLines.

Besides agents, resources, and transactions, the data model defines the output types of reports (Appendix C.1.3) the input types of contracts (Appendix C.1.4), and generic data definitions such as Address and OrderLine. The report types define the five mandatory reports mentioned earlier, and additional Inventory and TopNCustomers report types. The contract types define the two types of contracts for the bicycle company, namely Purchase and Sale.

3.2 Reports

Report specifications are divided into prelude functions (Appendix C.2.1), domain-specific prelude functions (Appendix C.2.2), internal reports (Appendix C.2.3), and external reports (Appendix C.2.4).

Prelude functions are utility functions that are independent of the custom data model. These functions are automatically added to all POETS instances, but they are included in the appendix for completeness. The prelude includes standard functions such as *filter*, but it also includes generators for accessing event log data such as *reports*. The event log generators provide access to data that has a lifecycle such as contracts or reports, compare Section 2.2.

Domain-specific prelude functions are utility functions that depend on the custom data model. The *itemsReceived* function, for example, computes a list of all items that have been delivered to the company, and it hence relies on the Delivery transaction type (*normaliseItems* and *isMe* are also defined in Appendix C.2.2):

```
itemsReceived : [Item]
itemsReceived = normaliseItems [is |
  tr ← transactionEvents,
  del : Delivery = tr.transaction,
  ¬(isMe del.sender) ∧ isMe del.receiver,
  is ← del.items]
```

Internal reports are reports that are needed either by clients of the system or by contracts. For instance, the *ContractTemplates* report is needed by clients of the system in order to instantiate contracts, and the *Me* report is needed by the two contracts, as we shall see in the following subsection. A list of internal reports, including a short description of what they compute, is summarised in Figure 11. Except for the *Me* report, all internal reports are independent from the custom data model.

External reports are reports that are expected to be rendered directly in clients of the system, but they may also be invoked by contracts. The external reports in our use case are the reports mentioned earlier, namely the income statement, the balance sheet, the cash flow statement, the list of unpaid invoices, and the VAT report. Moreover, we include reports for calulating the list of items in the inventory, and the list of top-n customers, respectively. We include the inventory report below as an example:

```
report : Inventory
report =
 let itemsSold' = map (λi → i{quantity = 0 − i.quantity}) itemsSold
 in
 −− The available items is the list of received items minus the
 −− list of reserved or sold items
```

Report	**Result**
Me	The special Me entity.
Entities	A list of all non-deleted entities.
EntitiesByType	A list of all non-deleted entities of a given type.
ReportNames	A list of names of all non-deleted reports.
ReportNamesByTags	A list of names of all non-deleted reports whose tags contain a given set and do not contain another given set.
ReportTags	A list of all tags used by non-deleted reports.
ContractTemplates	A list of names of all non-deleted contract templates.
ContractTemplatesByType	A list of names of all non-deleted contract templates of a given type.
Contracts	A list of all non-deleted contract instances.
ContractHistory	A list of previous transactions for a given contract instance.
ContractSummary	A list of meta data for a given contract instance.

Figure 11: Internal reports.

$$\mathsf{Inventory}\{\mathit{availableItems} = \mathit{normaliseItems}\ (\mathit{itemsReceived} +\!\!+\ \mathit{itemsSold}')\}$$

The value *itemsSold* is defined in the domain-specific prelude, similar to the value *itemsReceived*. But unlike *itemsReceived*, the computation takes into account that items can be reserved but not yet delivered. Hence when we check that items are in stock using the inventory report, we also take into account that some items in the inventory may have been sold, and therefore cannot be sold again.

The five standard reports are defined according to the specifications given by Henglein et al. [6, Section 2.1], but for simplicity we do not model fixed costs, depreciation, and fixed assets. We do, however, model multiple currencies, exemplified via Danish Kroner (DKK) and Euro (EUR). This means that financial reports, such as IncomeStatement, provide lists of values of type Money—one for each currency used.

3.3 Contracts

Contract specifications are divided into prelude functions (Appendix C.3.1), domain-specific prelude functions (Appendix C.3.2), and contract templates (Appendix C.3.3).

Prelude functions are utility functions similar to the report engine's prelude functions. They are independent from the custom data model, and are automatically added to all POETS instances. The prelude includes standard functions such as *filter*.

Domain-specific prelude functions are utility functions that depend on the custom data model. The *inStock* function, for example, checks whether the items described in a list of order lines are in stock, by querying the *Inventory* report (we assume that the item types are different for each line):

```
fun inStock lines =
  let inv = (reports.inventory ()).availableItems
  in
  all (λl → any (λi → (l.item).itemType ≡ i.itemType ∧
                           (l.item).quantity ≤ i.quantity) inv) lines
```

Contract templates describe the daily activities in the company, and in our μERP use case we only consider a purchase contract and a sales contract. The purchase contract is presented below:

```
name: purchase
type: Purchase
description: "Set up a purchase"

clause purchase(lines : [OrderLine])⟨me : ⟨Me⟩, vendor : ⟨Vendor⟩⟩ =
 ⟨vendor⟩ Delivery(sender s, receiver r, items i)
   where s ≡ vendor ∧ r ≡ me ∧ i ≡ map (λx → x.item) lines
   due within 1W
 then
 when IssueInvoice(sender s, receiver r, orderLines sl)
   where s ≡ vendor ∧ r ≡ me ∧ sl ≡ lines
   due within 1Y
 then
 payment(lines, vendor, 14D)⟨me⟩

clause payment(lines : [OrderLine], vendor : ⟨Vendor⟩, deadline : Duration)
                    ⟨me : ⟨Me⟩⟩ =
 if null lines then
   fulfilment
 else
   ⟨me⟩ BankTransfer(sender s, receiver r, money m)
     where s ≡ me ∧ r ≡ vendor ∧ checkAmount m lines
     due within deadline
     remaining newDeadline
   then
   payment(remainingOrderLines m lines, vendor, newDeadline)⟨me⟩

contract = purchase(orderLines)⟨me, vendor⟩
```

The contract describes a simple workflow, in which the vendor delivers items, possibly followed by an invoice, which in turn is followed by a bank transfer of the company. Note how the *me* parameter in the contract template body refers to the value from the domain-specific prelude, which in turn invokes the *Me* report. Note also how the *payment* clause template is recursively defined in order to accommodate for potentially different currencies. That is, the total payment is split up in as many bank transfers as there are currencies in the purchase.

The sales contract is presented below:

```
name: sale
type: Sale
description: "Set up a sale"

clause sale(lines : [OrderLine])⟨me : ⟨Me⟩, customer : ⟨Customer⟩⟩ =
 ⟨me⟩ IssueInvoice(sender s, receiver r, orderLines sl)
   where $s \equiv me \wedge r \equiv customer \wedge sl \equiv lines \wedge inStock\ lines$
   due within $1H$
 then
 payment(lines, me, $10m$)⟨customer⟩
 and
 ⟨me⟩ Delivery(sender s, receiver r, items i)
   where $s \equiv me \wedge r \equiv customer \wedge i \equiv map\ (\lambda x \to x.item)\ lines$
   due within $1W$
 then
 repair(map $(\lambda x \to x.item)$ lines, customer, $3M$)⟨me⟩

clause payment(lines : [OrderLine], me : ⟨Me⟩, deadline : Duration)
                    ⟨customer : ⟨Customer⟩⟩ =
 if null lines then
   fulfilment
 else
   ⟨customer⟩ Payment(sender s, receiver r, money m)
     where $s \equiv customer \wedge r \equiv me \wedge checkAmount\ m\ lines$
     due within deadline
     remaining newDeadline
   then
   payment(remainingOrderLines m lines, me, newDeadline)⟨customer⟩

clause repair(items : [Item], customer : ⟨Customer⟩, deadline : Duration)
                   ⟨me : ⟨Me⟩⟩ =
 when RequestRepair(sender s, receiver r, items i)
   where $s \equiv customer \wedge r \equiv me \wedge subset\ i\ items$
   due within deadline
   remaining newDeadline
 then
 ⟨me⟩ Repair(sender s, receiver r, items i')
   where $s \equiv me \wedge r \equiv customer \wedge i \equiv i'$
   due within $5D$
 and
 repair(items, customer, newDeadline)⟨me⟩

contract = sale(orderLines)⟨me, customer⟩
```

The contract describes a workflow, in which the company issues an invoice to the customer—but only if the items on the invoice are in stock. The issuing of invoice is followed by an immediate (within an hour) payment by the customer to the company, and a delivery of goods by the company within a week. Moreover, we also model the repair guarantee mentioned in the introduction.

3.4 Bootstrapping the System

The previous subsections described the specification code for μERP. Since data definitions, report specifications, and contract specifications are added to the system at run-time, μERP is instantiated by invoking the following sequence of services on an initially empty POETS instance:

1. Add data definitions in Appendix C.1 via *addDataDefs*.
2. Create a designated Me entity via *createEntity*.
3. Add report specifications via *addReport*.
4. Add contract specifications via *createTemplate*.

Hence, the event log will, conceptually, have the form (we write the value of the field internalTimeStamp before each event):

t_1: AddDataDefs{defs = `"ResourceType is ..."`}

t_2: CreateEntity{ent = e_1, recordType = `"Me"`, data = Me}

t_3: CreateReport{name = `"Me"`, description = `"Returns the ..."`,
code = `"name: Me\n ..."`, tags = [`"internal"`,`"entity"`]}

⋮

t_i: CreateReport{name = `"TopNCustomers"`, description = `"A list ..."`,
code = `"name: TopNCustomers\n ..."`,
tags = [`"external"`,`"financial"`,`"crm"`]}

t_{i+1}: CreateContractDef{name = `"Purchase"`, recordType = `"Purchase"`,
code = `"name: purchase\n ..."`, description = `"Set up ..."`}

t_{i+2}: CreateContractDef{name = `"Sale"`, recordType = `"Sale"`,
code = `"name: sale\n ..."`, description = `"Set up a sale"`}

for some increasing timestamps $t_1 < t_2 < \ldots < t_{i+2}$. Note that the entity value e_1 of the CreateEntity event is automatically generated by the entity store, as described in Section 2.3.

Following these operations, the system is operational. That is, (i) customers and vendors can be managed via *createEntity*, *updateEntity*, and *deleteEntity*, (ii) contracts can be instantiated, updated, concluded, and inspected via *createContract*, *updateContract*, *concludeContract*, and *getContract* respectively, (iii) transactions can be registered via *registerTransaction*, and (iv) reports can be queried via *queryReport*.

For example, if a sale is initiated with a new customer John Doe, starting at time t, then the following events will be added to the event log:

t_{i+3}: CreateEntity{ent = e_2, recordType = "Customer", data = Customer{
name = "John Doe", address = Address{
string = "Universitetsparken 1", country = Denmark}}}

t_{i+4}: CreateContract{contractId = 0, contract = Sale{
startDate = t, templateName = "sale", customer = e_2,
orderLines = [OrderLine{
item = Item{itemType = Bicycle{model = "Avenue"}, quantity = 1.0},
unitPrice = Money{currency = DKK, amount = 4000.0},
vatPercentage = 25.0}]}}

That is, first the customer entity is created, and then we can instantiate a new sales contract. In this particular sale, one bicycle of the model "Avenue" is sold at a unit price of 4000 DKK, with an additional VAT of 25 percent. Note that the contract id 0 of the CreateContract is automatically generated and that the start time t is explicitly given in the CreateContract's startDate field independent from the internalTimeStamp field.

Following the events above, if the contract is executed successfully, events of type IssueInvoice, Delivery, and Payment will persisted in the event log with appropriate values—in particular, the payment will be 5000 DKK.

4 Implementation Aspects

In this section we briefly discuss some of the implementation techniques used in our implementation of POETS. POETS is implemented in Haskell [9], and the logical structure of the implementation reflects the diagram in Figure 2, that is each component is implemented as a separate Haskell module.

4.1 External Interface

The external interface to the POETS system is implemented in a separate Haskell module. We currently use Thrift [15] for implementing the communication layer between the server and its clients, but other communication layers can in principle be used. Changing the communication layer will only require a change in one module.

Besides offering an abstract, light-weight interface to communication, Thrift enables type-safe communication. The types and services of the server are specified in a language-independent description language, from which Haskell code is generated (or code in other languages for the clients). For example, the external interface to querying a report can be specified as follows:

```
Value queryReport(
  1 : string name      // name of the report to execute
  2 : list<Value> args // input arguments
) throws (
  1 : ReportNotFoundException notFound
  2 : RunTimeException runtime
  3 : TypeException type
)
```

From this specification, Thrift generates the Haskell code for the server interface, and implementing the interface amounts to supplying a function of the type $String \to [\,Value\,] \to IO\ \ Value$—namely the query function.

4.2 Domain-Specific Languages

The main ingredient of the POETS implementation is the implementation of the domain-specific languages. What is interesting in that respect—compared to implementations of domain-specific languages in isolation of each other—is the common core shared by the languages, in particular types and values.

In order to reuse and extend the structure of types and values in the report language and the contract language, we make use of the *compositional data types* [2] library. Compositional data types take the *data types as fixed points* [11] view on abstract syntax trees (ASTs), namely a separation of the recursive structure of ASTs from their signatures. As an example, we define the signatures of types from Section 2.1.1 as follows:

```
type RecordName      = String
data TypeConstant a = TBool | TInt | ···
data TypeRecord a   = TRecord RecordName
data TypeList a     = TList a
data TypeEnt a      = TEnt RecordName
```

The signature for the types of the data model is then obtained by combining the individual signatures above $TSig = TypeConstant$:+: $TypeRecord$:+: $TypeList$:+: $TypeEnt$, where $(:+:) :: (* \to *) \to (* \to *) \to * \to *$ is the sum of two functors. Finally, the data type for ASTs of types can be defined by tying the recursive knot $T = Term\ TSig$, where $Term :: (* \to *) \to *$ is the functor fixed point.

Recursive functions over ASTs are defined as type classes, with one instance per atomic signature. For instance, a pretty printer for types can be defined as follows:

```
class Functor f ⇒ Render f where
  render :: f String → String
instance Render TypeConstant where
  render TInt  = "Int"
  render TBool = "Bool"
  ···
instance Render TypeRecord where
  render (TRecord r) = r
instance Render TypeList where
```

```
    render (TList τ) = "[" ++ τ ++ "]"
  instance Render TypeEnt where
    render (TEnt r) = "<" ++ r ++ ">"
```

and pretty printing of terms is subsequently obtained by lifting the *render* algebra to a catamorphism, that is a function of type *Render f* ⇒ *Term f* → *String*.

Extendability The first benefit of the approach above is that we can extend the signature for types to fit, for example, the contract language as in Figure 9:

```
type TypeVar           = String
data TypeUnit a        = TUnit
data TypeVar a         = TVar TypeVar
data TypeFunction a    = TFunction a a
```

Extending the pretty printer amounts to only providing the new cases:

```
instance render TypeUnit where
  render TUnit = "()"
instance render TypeVar where
  render (TVar α) = α
instance render TypeFunction where
  render (TFunction τ1 τ2) = τ1 ++ " -> " ++ τ2
```

A similar modular encoding is used for the language of values:

```
data Value a = VInt Int | VBool Bool | VString String | ···
```

and the signature of expressions in the contract language of Figure 9 can be obtained by providing the extensions compared to the language of values:

```
type Var = String
data Exp a = EVar Var | ELambda Var a | EApply a a | ···
```

That is, *Term* (*Exp* :+: *Value*) represents the type of ASTs for expressions of the contract language. Reusing the signature for (core) values means that the values of Section 2.1.2, which are provided as input to the system for instance in the *registerTransaction* function, can be automatically coerced to the richer language of expressions. That is, values of type *Term Value* can be readily used as values of type *Term* (*Exp* :+: *Value*), without explicit copying or translation.

Notice the difference in the granularity of (core) value signatures and (core) type signatures: types are divided into three signatures, whereas values are in one signature. The rule of thumb we apply is to divide signatures only when a function needs the granularity. For instance, the type inference algorithm used in the report language and the contract language implements a simplification procedure [5], which reduces type constraints to *atomic* type constraints. In order to guarantee this transformation invariant statically, we hence need a signature of atomic types, namely *TypeConstant* :+: *TypeVar*, which prompts the finer granularity on types.

Syntactic sugar Besides enabling a common core of ASTs and functions on them, compositional data type enable AST transformations where the invariant of the transformation is witnessed by the type. Most notably, desugaring can be implemented by providing a signature for syntactic sugar:

```
data ExpSug a = ELet Var a a | ···
```

as well as a transformation to the core signature:

```
instance Desugar ExpSug where
  desugar (ELet x e1 e2) = ELam x e2 ‘EApp‘ e1
  ···
```

This approach yields a desugaring function of the type *Term* (*ExpSug* :+: *Exp* :+: *Value*) $\to$ *Term* (*Exp* :+: *Value*), which witnesses that the syntactic sugar has indeed been removed.

Moreover, since we define the desugaring translation in the style of a *term homomorphism* [2], we automatically get a lifted desugaring function that propagates AST annotations, such as source code positions, to the desugared term. This means, for instance, that type error messages can provide detailed source position information also for terms that originate from syntactic sugar.

5 Conclusion

We have presented an extended and generalised version of the POETS architecture [6], which we have fully implemented. We have presented domain-specific languages for specifying the data model, reports, and contracts of a POETS instance, and we have demonstrated an application of POETS in a small use case. The use case demonstrates the conciseness of our approach—Appendix C contains the complete source needed for a running system—as well as the domain-orientation of our specification languages. We believe that non-programmers should be able to read and understand the data model of Appendix C.1, to some extent the contract specifications of Appendix C.3.3, and to a lesser extent the reports of Appendix C.2 (after all, reports describe computations).

5.1 Future Work

With our implementation and revision of POETS we have only taken the first steps towards a software system that can be used in practice. In order to properly verify our hypothesis that POETS is practically feasible, we want to conduct a larger use case in a live, industrial setting. Such use case will both serve as a means of testing the technical possibilities of POETS, that is whether we can model and implement more complex scenarios, as well as a means of testing our hypothesis that the use of domain-specific languages shortens the gap between requirements and implementation.

Expressivity As mentioned above, a larger and more realistic use case is needed in order to fully evaluate POETS. In particular, we are interested in investigating whether the data model, the report language, and the contract language have sufficient expressivity. For instance, a possible extension of the data model is to introduce finite maps. Such extension will, for example, simplify the reports from our μERP use case that deal with multiple currencies. Moreover, finite maps will enable a modelling of resources that is closer in structure to that of Henglein et al. [6].

Another possible extension is to allow types as values in the report language. For instance, the *EntitiesByType* report in Appendix C.2.3 takes a string representation of a record type, rather than the record type itself. Hence the function cannot take subtypes into account, that is if we query the report with input A, then we only get entities of declared type A and not entities of declared subtypes of A.

Rules A rule engine is a part of our extended architecture (Figure 2), however it remains to be implemented. The purpose of the rule engine is to provide rules—written in a separate domain-specific language—that can constrain the values that are accepted by the system. For instance, a rule might specify that the items list of a Delivery transaction always be non-empty.

More interestingly, the rule engine will enable values to be *inferred* from the rules in the engine. For instance, a set of rules for calculating VAT will enable the field vatPercentage of an OrderLine to be inferred automatically in the context of a Sale record. That is, based on the information of a sale and the items that are being sold, the VAT percentage can be calculated automatically for each item type.

The interface to the rule engine will be very simple: A record value, as defined in Section 2.1.2, with zero or more *holes* is sent to the engine, and the engine will return either (i) an indication that the record cannot possibly fulfil the rules in the engine, or (ii) a (partial) substitution that assigns inferred values to (some of) the holes of the value as dictated by the rules. Hence when we, for example, instantiate the sale of a bicycle in Section 3.4, then we first let the rule engine infer the VAT percentage before passing the contract meta data to the contract engine.

Forecasts A feature of the contract engine, or more specifically of the reduction semantics of contract instances, is the possibility to retrieve the state of a running contract at any given point in time. The state is essentially the AST of a contract clause, and it describes what is currently expected in the contract, as well as what is expected in the future.

Analysing the AST of a contract enables the possibility to do *forecasts*, for instance to calculate the expected outcome of a contract or the items needed for delivery within the next week. Forecasts are, in some sense, dual to reports. Reports derive data from transactions, that is facts about what has previously happened. Forecasts, on the other hand, look into the future, in terms of calculations over running contracts. We have currently implemented a single forecast, namely a forecast that lists the set of immediately expected transactions for a given contract. A more ambitious approach is to devise (yet another) language for writing forecasts, that is functions that operate on contract ASTs.

Practicality In order to make POETS useful in practice, many features are still missing. However, we see no inherent difficulties in adding them to POETS compared to traditional ERP architectures. To mention a few: (i) security, that is authorisation, users, roles, etc.; (ii) module systems for the report language and contract language, that is better support for code reuse; and (iii) check-pointing of a running system, that is a dump of the memory of a running system, so the event log does not have to be replayed from scratch when the system is restarted.

Acknowledgements We are grateful to Fritz Henglein for many fruitful discussions and for convincing us of the *POETS approach* in the first place. Morten Ib Nielsen and Mikkel Jønsson Thomsen both contributed to our implementation and design of POETS, for which we are thankful. Lastly, we thank the participants of the DIKU course "POETS Summer of Code" for valuable input.

References

[1] Jesper Andersen, Ebbe Elsborg, Fritz Henglein, Jakob Simonsen, and Christian Stefansen. Compositional specification of commercial contracts. *International Journal on Software Tools for Technology Transfer (STTT)*, 8:485–516, 2006.

[2] Patrick Bahr and Tom Hvitved. Compositional Data Types. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming*, pages 83–94, New York, NY, USA, 2011. ACM.

[3] Arthur J. Bernstein and Michael Kifer. *Databases and Transaction Processing: An Application-Oriented Approach.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 2001.

[4] Norbert Fuchs, Kaarel Kaljurand, and Tobias Kuhn. Attempto Controlled English for Knowledge Representation. In *Reasoning Web*, pages 104–124. Springer Berlin / Heidelberg, 2008.

[5] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73(2):155–175, 1990.

[6] Fritz Henglein, Ken Friis Larsen, Jakob Grue Simonsen, and Christian Stefansen. POETS: Process-oriented event-driven transaction systems. *Journal of Logic and Algebraic Programming*, 78(5):381–401, May 2009.

[7] Tom Hvitved, Felix Klaedtke, and Eugen Zălinescu. A Trace-based Model for Multiparty Contracts. *Journal of Logic and Algebraic Programming*, 2011. To appear.

[8] Mikkel Jønsson Thomsen. Using Controlled Natural Language for specifying ERP Requirements. Master's thesis, University of Copenhagen, Department of Computer Science, 2010.

[9] Simon Marlow. *Haskell 2010 Language Report*, 2010.

[10] William E. McCarthy. The REA Accounting Model: A Generalized Framework for Accounting Systems in a Shared Data Environment. *The Accounting Review*, LVII(3):554–578, 1982.

[11] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 124–144, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[12] Michael Nissen and Ken Friis Larsen. FunSETL — Functional Reporting for ERP Systems. In Olaf Chitil, editor, *Implementation and Application of Functional Languages, 19th International Symposium, IFL 2007*, pages 268–289, 2007.

[13] Atsushi Ohori. A Polymorphic Record Calculus and Its Compilation. *ACM Trans. Program. Lang. Syst.*, 17:844–895, November 1995.

[14] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 2002.

[15] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation. Technical report, Facebook, 156 University Ave, Palo Alto, CA, 2007.

[16] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(04):461–493, 1992.

[17] Jerry J. Weygandt, Donald E. Kieso, and Paul D. Kimmel. *Financial Accounting, with Annual Report.* Wiley, 2004.

A Predefined Ontology

A.1 Data

Data is abstract.

A.2 Event

Event is abstract.
Event has a Timestamp
    called internalTimeStamp.

# Add data definitions to the system
AddDataDefs is an Event.
AddDataDefs has a String called defs.

# Events associated with entities
EntityEvent is an Event.
EntityEvent is abstract.
EntityEvent has a Data entity called ent.

# Put entity event
PutEntity is an EntityEvent.
PutEntity has Data.
PutEntity is abstract.

# Create entity event
CreateEntity is a PutEntity.
CreateEntity has a String called recordType.

# Update entity event
UpdateEntity is a PutEntity.

# Delete entity event
DeleteEntity is an EntityEvent.

# Events associated with a report definition
ReportEvent is an Event.
ReportEvent has a String called name.

# Put report definition event
PutReport is a ReportEvent.
PutReport is abstract.
PutReport has a String called code.
PutReport has a String called description.
PutReport has a list of String called tags.

# Create report definition event
CreateReport is a PutReport.

# Update report definition event
UpdateReport is a PutReport.

# Delete report definition event
DeleteReport is a ReportEvent.

# Events associated with a contract template
ContractDefEvent is an Event.
ContractDefEvent has a String called name.

# Put contract template event
PutContractDef is a ContractDefEvent.
PutContractDef is abstract.
PutContractDef has a String called recordType.
PutContractDef has a String called code.
PutContractDef has a String called description.

# Create contract template event
CreateContractDef is a PutContractDef.

# Update contract template event
UpdateContractDef is a PutContractDef.

# Delete contract template event
DeleteContractDef is a ContractDefEvent.

# Events associated with a contract
ContractEvent is an Event.
ContractEvent is abstract.
ContractEvent has an Int called contractId.

# Put contract event
PutContract is a ContractEvent.
PutContract has a Contract.
PutContract is abstract.

# Create contract event
CreateContract is a PutContract.

# Update contract event
UpdateContract is a PutContract.

# Conclude contract event
ConcludeContract is a ContractEvent.

# Transaction super class
TransactionEvent is a ContractEvent.
TransactionEvent has a Timestamp.
TransactionEvent has a Transaction.

A.3 Transaction

Transaction is abstract.

A.4 Report

Report is abstract.

A.5 Contract

Contract is abstract.
Contract has a Timestamp called startDate.
Contract has a String called templateName.

B Static and Dynamic Semantics of the Report Language

B.1 Types, Type Constraints and Type Schemes

The following grammar describes the type expressions that are used in the report language:

$$\begin{aligned}\tau ::=\ & r \mid \alpha \mid \textbf{Bool} \mid \textbf{Int} \mid \textbf{Real} \mid \textbf{Char} \mid \textbf{Timestamp} \mid \textbf{Duration} \\ & \mid \textbf{DurationTimestamp} \mid [\tau] \mid \langle r \rangle \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid (\tau_1, \tau_2) \mid ()\end{aligned}$$

where r ranges over record names and α over type variables.

The report language is polymorphically typed and permits to put constraints on types, for example, subtyping constraints. The language of type constraints is defined as follows:

$$C ::= \tau_1 <: \tau_2 \mid \tau_1.f : \tau_2 \mid \mathsf{Eq}(\tau) \mid \mathsf{Ord}(\tau)$$

Intuitively, these constraints can be interpreted as follows:

- A *subtype constraint* of the form $\tau_1 <: \tau_2$ requires τ_1 to be a subtype of τ_2,
- a *field constraint* of the form $\tau_1.f : \tau_2$ requires τ_1 to be a record type containing a field f of type τ_2,
- an *equality constraint* of the form $\mathsf{Eq}(\tau)$ requires the type τ to have an equality predicate $\equiv$ defined on it, and
- an *order constraint* of the form $\mathsf{Ord}(\tau)$ requires the type τ to have order predicates ($<$, $\leq$) defined on it.

In order to accommodate for the polymorphic typing, we have to move from types to *type schemes*. Type schemes are of the form $\forall\overline{\alpha}.\overline{C} \Rightarrow \tau$, that is, a type with a universal quantification over a sequence of type variables, restricted by a sequence of constraints. We abbreviate $\forall\langle\rangle.\overline{C} \Rightarrow \tau$ by writing $\overline{C} \Rightarrow \tau$, and $\langle\rangle \Rightarrow \tau$ by τ. The *universal closure* of a type scheme $\overline{C} \Rightarrow \tau$, that is, $\forall\overline{\alpha}.\overline{C} \Rightarrow \tau$ for $\overline{\alpha}$ the free variables $\mathsf{fv}(\overline{C}, \tau)$ in $\overline{C}$ and τ, is abbreviated by $\forall\overline{C} \Rightarrow \tau$.

B.2 Built-in Symbols

In the following we give an overview of the constants provided by the language. Along with each constant c we will associate a designated type scheme σ_c.

One part of the set of constants consists of literals: Numeric literals $\mathbb{R}$, Boolean literals $\{\textbf{True}, \textbf{False}\}$, character literals $\{\texttt{'a'}, \texttt{'b'}, \ldots\}$, and string literals. Each literal is associated with its obvious type: **Int** (respectively **Real**), **Bool**, **Char**, respectively **String**. Moreover, we also have entity values $\langle r, e\rangle$ of type $\langle r\rangle$ with e a unique identifier.

In the following we list the remaining built-in constants along with their respective type schemes. Many of the given constant symbols are used as mixfix operators. This is indicated by placeholders $\_$. For example a binary infix operator $\circ$ is then written as a constant $\_\circ\_$. For a constant c we write $c : \overline{C} \Rightarrow \tau$ in order to indicate the type scheme $\sigma_c = \forall \overline{C} \Rightarrow \tau$ assigned to c.

$$\begin{array}{rll}
\_\circ\_ : & \alpha <: \mathbf{Real} \Rightarrow \alpha \to \alpha \to \alpha & \forall \circ \in \{+, -, *\} \\
\_/\_ : & \mathbf{Real} \to \mathbf{Real} \to \mathbf{Real} & \\
\_\equiv\_ : & \mathsf{Eq}(\alpha) \Rightarrow \alpha \to \alpha \to \mathbf{Bool} & \\
\_\circ\_ : & \mathsf{Ord}(\alpha) \Rightarrow \alpha \to \alpha \to \mathbf{Bool} & \forall \circ \in \{>, \geq, <, \leq\} \\
\_\circ\_ : & \alpha <: \mathbf{DurationTimestamp} \Rightarrow \alpha \to \mathbf{Duration} \to \alpha & \forall \circ \in \{\langle + \rangle, \langle - \rangle\}
\end{array}$$

$$\begin{array}{rl}
r\ \{f_1 = \_, \ldots, f_n = \_\} : & \tau_1 \to \ldots \tau_n \to r \qquad \text{where } \rho(r) = \{(f_1, \tau_1), \ldots, (f_n, \tau_n)\} \\
\_.f : & \alpha.f : \beta \Rightarrow \alpha \to \beta \\
\_\ \{f_1 = \_, \ldots, f_n = \_\} : & \alpha.f_1 : \alpha_1, \ldots, \alpha.f_n : \alpha_n \Rightarrow \alpha \to \alpha_1 \to \ldots \to \alpha_n \to \alpha
\end{array}$$

$$\begin{array}{rll}
\neg : & \mathbf{Bool} \to \mathbf{Bool} & \\
\_\circ\_ : & \mathbf{Bool} \to \mathbf{Bool} \to \mathbf{Bool} & \forall \circ \in \{\wedge, \vee\} \\
\mathbf{if}\ \_\ \mathbf{then}\ \_\ \mathbf{else}\ \_ : & \mathbf{Bool} \to \alpha \to \alpha \to \alpha &
\end{array}$$

$$\begin{array}{rl}
[\,] : & [\alpha] \\
\_\#\_ : & \alpha \to [\alpha] \to [\alpha] \\
\mathbf{fold} : & \mathsf{Eq}(\beta) \Rightarrow (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta
\end{array}$$

$$\begin{array}{rl}
() : & () \\
(\_, \_) : & \alpha \to \beta \to (\alpha, \beta) \\
\mathbf{Inl} : & \alpha \to \alpha + \beta \\
\mathbf{Inr} : & \beta \to \alpha + \beta \\
\mathbf{case} : & \alpha + \beta \to (\alpha \to \gamma) \to (\beta \to \gamma) \to \gamma \\
\_.1 : & (\alpha, \beta) \to \alpha \\
\_.2 : & (\alpha, \beta) \to \beta
\end{array}$$

$$\begin{array}{rl}
\_! : & \langle r \rangle \to r \\
\_@ : & \langle r \rangle \to r
\end{array}$$

$$\begin{array}{rl}
\langle\langle \_ - \_ - \_ \quad \_ : \_ : \_ \rangle\rangle : & \underbrace{\mathbf{Int} \to \cdots \to \mathbf{Int}}_{6\times} \to \mathbf{Timestamp} \\
\langle\langle \_\ s, \_\ min, \_\ h, \_\ d, \_\ w, \_\ mon, \_\ y \rangle\rangle : & \underbrace{\mathbf{Int} \to \cdots \to \mathbf{Int}}_{7\times} \to \mathbf{Duration}
\end{array}$$

$$\textbf{error} : \textbf{String} \to \alpha$$

We assume that there is always defined a record type Event which is the type of an event stored in the central *event log* of the system. The list of all events in the event log can be accessed by the following constant:

$$\textbf{events} : [\mathsf{Event}]$$

When considering built-in constants, we also distinguish between *defined functions* f and *constructors* F. Constructors are the constants $\langle\langle \_ - \_ - \_ \;\; \_ : \_ : \_ \rangle\rangle$, $\langle\langle \_ \, s, \_ \, min, \_ \, h, \_ \, d, \_ \, w, \_ \, mon, \_ \, y \rangle\rangle$, $r\ \{f_1 = \_, \dots, f_n = \_\}$, $\#$, $[]$, $()$, $(\_, \_)$, **Inl**, **Inr** and **error** as well as all literals. The remaining constants are defined functions.

Derived from its type scheme we can also assign an *arity* $\mathsf{ar}(c)$ to each constant c by defining $\mathsf{ar}(c)$ as the largest n such that $\sigma_c = \forall\overline{\alpha}.\overline{C} \Rightarrow \tau_1 \to \tau_2 \to \dots \to \tau_{n+1}$

B.3 Type System

Before we can present the type system of the report language, we have to give the rules for the type constraints. To this end we extend the subtyping judgement $\mathcal{R} \vdash \tau_1 <: \tau_2$ for values from Figure 4. The constraint entailment judgement $\mathcal{R}, \mathcal{C} \Vdash C$ states that a constraint C follows from the set of constraints $\mathcal{C}$ and the record typing environment $\mathcal{R}$.

The type constraint entailment judgement $\mathcal{R}, \mathcal{C} \Vdash C$ is straightforwardly extended to sequences of constraints $\overline{C}$. We define that $\mathcal{R}, \mathcal{C} \Vdash C_1, \dots, C_n$ iff $\mathcal{R}, \mathcal{C} \Vdash C_i$ for all $1 \le i \le n$.

The type system of the report language is a straightforward polymorphic lambda calculus extended with type constraints. The typing judgement for the report language is written $\mathcal{R}, \mathcal{C}, \Gamma \vdash e : \sigma$, where $\mathcal{R}$ is a record typing environment, $\mathcal{C}$ a set of type constraints, Γ a type environment, e an expression and σ a type scheme. The inference rules for this judgement are given in Figure 13.

A typing $\mathcal{R}, \mathcal{C}', \Gamma' \vdash e : \tau'$ is an instance of $\mathcal{R}, \mathcal{C}, \Gamma \vdash e : \tau$ iff there is a substitution S such that $\Gamma' \supseteq \Gamma S$, $\tau' = \tau S$, and $\mathcal{R}, \mathcal{C}' \Vdash \mathcal{C} S$. Deriving from that we say that the type scheme $\sigma' = \forall\overline{\alpha}'.\overline{C}' \Rightarrow \tau'$ is an instance of $\sigma = \forall\overline{\alpha}.\overline{C} \Rightarrow \tau$, written $\sigma' < \sigma$, iff there is a substitution S with $\mathrm{dom}(S) = \alpha$ such that $\tau' = \tau S$ and $\mathcal{R}, \mathcal{C}' \Vdash \mathcal{C} S$.

Top-level function definitions are of the form

$$f\ x_1\ \dots\ x_n = e$$

and can be preceded by an explicit type signature declaration of the form $f : \sigma$.

Depending on whether an explicit type signature is present, the following inference rules define the typing of top-level function definitions:

$$\frac{\mathcal{R}, \mathcal{C} \cup \overline{C}, \Gamma \cup \{x_1 : \tau_1, \dots, x_n : \tau_n\} \vdash e : \tau \qquad \overline{\alpha} \notin \mathsf{fv}(\mathcal{C}) \cup \mathsf{fv}(\Gamma)}{\mathcal{R}, \mathcal{C}, \Gamma \vdash f\ x_1\ \dots\ x_n = e : \forall\overline{\alpha}.\overline{C} \Rightarrow \tau_1 \to \dots \to \tau_n \to \tau}\ \text{(Fun)}$$

$$\frac{C \in \mathcal{C}}{\mathcal{R}, \mathcal{C} \Vdash C}\;(\text{Hyp}) \qquad \frac{r_1 \leq r_2}{(R, A, F, \rho, \leq), \mathcal{C} \Vdash r_1 <: r_2}\;(<:\text{ Rec})$$

$$\frac{}{\mathcal{R}, \mathcal{C} \Vdash \tau <: \tau}\;(<:\text{ Refl}) \qquad \frac{\mathcal{R}, \mathcal{C} \Vdash \tau_1 <: \tau_2 \qquad \mathcal{R}, \mathcal{C} \Vdash \tau_2 <: \tau_3}{\mathcal{R}, \mathcal{C} \Vdash \tau_1 <: \tau_3}\;(<:\text{ Trans})$$

$$\frac{\mathcal{R}, \mathcal{C} \Vdash \tau_1 <: \tau_2 \qquad \mathcal{R}, \mathcal{C} \Vdash \tau_3 <: \tau_4}{\mathcal{R}, \mathcal{C} \Vdash \tau_2 \to \tau_3 <: \tau_1 \to \tau_4}\;(<:\text{ Fun}) \qquad \frac{\mathcal{R}, \mathcal{C} \Vdash \tau_1 <: \tau_2}{\mathcal{R}, \mathcal{C} \Vdash [\tau_1] <: [\tau_2]}\;(<:\text{ List})$$

$$\frac{\mathcal{R}, \mathcal{C} \Vdash \tau_1 <: \tau_2 \qquad \mathcal{R}, \mathcal{C} \Vdash \tau_3 <: \tau_4}{\mathcal{R}, \mathcal{C} \Vdash \tau_1 + \tau_3 <: \tau_2 + \tau_4}\;(<:\text{ Sum})$$

$$\frac{\mathcal{R}, \mathcal{C} \Vdash \tau_1 <: \tau_2 \qquad \mathcal{R}, \mathcal{C} \Vdash \tau_3 <: \tau_4}{\mathcal{R}, \mathcal{C} \Vdash (\tau_1, \tau_3) <: (\tau_2, \tau_4)}\;(<:\text{ Prod})$$

$$\frac{}{\mathcal{R}, \mathcal{C} \Vdash \mathbf{Int} <: \mathbf{Real}}\;(<:\text{ Num})$$

$$\frac{}{\mathcal{R}, \mathcal{C} \Vdash \mathbf{Timestamp} <: \mathbf{DurationTimestamp}}\;(<:\text{ Timestamp})$$

$$\frac{}{\mathcal{R}, \mathcal{C} \Vdash \mathbf{Duration} <: \mathbf{DurationTimestamp}}\;(<:\text{ Duration})$$

$$\frac{(f, \tau) \in \rho(r)}{(R, A, F, \rho, \leq), \mathcal{C} \Vdash r.f : \tau}\;(\text{Field})$$

$$\frac{\mathcal{R}, \mathcal{C} \Vdash \tau_1.f : \tau_2 \qquad \mathcal{R}, \mathcal{C} \Vdash \tau_1' <: \tau_1}{\mathcal{R}, \mathcal{C} \Vdash \tau_1'.f : \tau_2}\;(\text{Field Prop})$$

$$\frac{\tau \in \{\mathbf{Bool}, \mathbf{Int}, \mathbf{Real}, \mathbf{Char}, \mathbf{Duration}, \mathbf{Timestamp}, \mathbf{DurationTimestamp}\}}{\mathcal{R}, \mathcal{C} \Vdash \mathsf{Ord}(\tau)}\;(\text{Ord Base})$$

$$\frac{\mathcal{R}, \mathcal{C} \Vdash \mathsf{Ord}(\tau)}{\mathcal{R}, \mathcal{C} \Vdash \mathsf{Eq}(\tau)}\;(\text{Eq Ord}) \qquad \frac{r \in R}{(R, A, F, \rho, \leq), \mathcal{C} \Vdash \mathsf{Eq}(r)}\;(\text{Eq Rec})$$

$$\frac{F \in \{(\cdot,\cdot), +, [\cdot], \langle\cdot\rangle\} \qquad P \in \{\mathsf{Ord}(\cdot), \mathsf{Eq}(\cdot)\} \qquad \forall 1 \leq i \leq n\colon\ \mathcal{R}, \mathcal{C} \Vdash P(\tau_i)}{\mathcal{R}, \mathcal{C} \Vdash P(F(\tau_1, \ldots, \tau_n))}\;(P\ F)$$

Figure 12: Type constraint entailment $\mathcal{R}, \mathcal{C} \Vdash C$.

$$\frac{x : \sigma \in \Gamma}{\mathcal{R}, \mathcal{C}, \Gamma \vdash x : \sigma}\ (\text{Var}) \qquad \frac{}{\mathcal{R}, \mathcal{C}, \Gamma \vdash c : \sigma_c}\ (\text{Const})$$

$$\frac{\mathcal{R}, \mathcal{C}, \Gamma \vdash e : \tau \qquad \mathcal{C} \Vdash \tau <: \tau'}{\mathcal{R}, \mathcal{C}, \Gamma \vdash e : \tau'}\ (\text{Sub}) \qquad \frac{\mathcal{R}, \mathcal{C}, \Gamma \cup \{x : \tau\} \vdash e : \tau'}{\mathcal{R}, \mathcal{C}, \Gamma \vdash \lambda x \to e : \tau \to \tau'}\ (\text{Abs})$$

$$\frac{\mathcal{R}, \mathcal{C}, \Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \mathcal{R}, \mathcal{C}, \Gamma \vdash e_2 : \tau_1}{\mathcal{R}, \mathcal{C}, \Gamma \vdash e_1\ e_2 : \tau_2}\ (\text{App})$$

$$\frac{\mathcal{R}, \mathcal{C}, \Gamma \vdash e_1 : \sigma \qquad \mathcal{R}, \mathcal{C}, \Gamma \cup \{x : \sigma\} \vdash e_2 : \tau}{\mathcal{R}, \mathcal{C}, \Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau}\ (\text{Let})$$

$$\frac{\begin{array}{cc}\mathcal{R}, \mathcal{C}, \Gamma \vdash e : r' & \mathcal{R}, \mathcal{C}, \Gamma \cup \{x : r\} \vdash e_1 : \tau \\ \mathcal{R}, \mathcal{C} \Vdash r <: r' & \mathcal{R}, \mathcal{C}, \Gamma \cup \left\{x : r'\right\} \vdash e_2 : \tau\end{array}}{\mathcal{R}, \mathcal{C}, \Gamma \vdash \textbf{type}\ x = e\ \textbf{of}\ \{r \to e_1; \_ \to e_2\} : \tau}\ (\text{Type Of})$$

$$\frac{\begin{array}{cc}\mathcal{R}, \mathcal{C}, \Gamma \vdash e : \langle r' \rangle & \mathcal{R}, \mathcal{C}, \Gamma \cup \{x : \langle r \rangle\} \vdash e_1 : \tau \\ \mathcal{R}, \mathcal{C} \Vdash r <: r' & \mathcal{R}, \mathcal{C}, \Gamma \cup \left\{x : \langle r' \rangle\right\} \vdash e_2 : \tau\end{array}}{\mathcal{R}, \mathcal{C}, \Gamma \vdash \textbf{type}\ x = e\ \textbf{of}\ \{\langle r \rangle \to e_1; \_ \to e_2\} : \tau}\ (\text{Type Of Ref})$$

$$\frac{\mathcal{R}, \mathcal{C} \cup \overline{C}, \Gamma \vdash e : \tau \qquad \overline{\alpha} \notin \mathsf{fv}(\mathcal{C}) \cup \mathsf{fv}(\Gamma)}{\mathcal{R}, \mathcal{C}, \Gamma \vdash e : \forall \overline{\alpha}.\overline{C} \Rightarrow \tau}\ (\forall\ \text{Intro})$$

$$\frac{\mathcal{R}, \mathcal{C}, \Gamma \vdash e : \forall \overline{\alpha}.\overline{C} \Rightarrow \tau' \qquad \mathcal{R}, \mathcal{C} \Vdash \overline{C}\,[\overline{\alpha}/\overline{\tau}]}{\mathcal{R}, \mathcal{C}, \Gamma \vdash e : \tau'\,[\overline{\alpha}/\overline{\tau}]}\ (\forall\ \text{Elim})$$

Figure 13: Type inference rules for the report language.

$$\frac{\mathcal{R}, \mathcal{C}, \Gamma \vdash f\ x_1\ \ldots\ x_n = e : \sigma \qquad \sigma' < \sigma}{\mathcal{R}, \mathcal{C}, \Gamma \vdash f \colon \sigma'; f\ x_1\ \ldots\ x_n = e : \sigma'}\ (\text{Fun'})$$

B.4 Operational Semantics

In order to simplify the presentation of the operational semantics we assign to each constant c of the language its set of *strict argument positions* $\mathsf{strict}(c) \subseteq \{1, \ldots, \mathsf{ar}(c)\}$:

$$\begin{array}{rll}
\mathsf{strict}(\_ \circ \_) = \{1, 2\} & & \text{for all binary operators } \circ \neq \# \\
\mathsf{strict}(c) = \{1\} & & \forall c \in \{\neg, \textbf{if}\ \_\ \textbf{then}\ \_\ \textbf{else}\ \_, \textbf{case}, \textbf{error}, \_@, \_!\} \\
\mathsf{strict}(\_.f) = \{1\} & & \\
\mathsf{strict}(\_\ \{\overline{f_i = e_i}\}) = \{1\} & & \\
\mathsf{strict}(\_.i) = \{1\} & &
\end{array}$$

For all other constraints c for which the above equations do not apply $\mathsf{strict}(c)$ is defined as the empty set $\emptyset$.

Values form a subset of expressions which are fully evaluated at the top-level. Such expressions are also said to be in *weak head normal form* (*whnf*). An expression is in weak head normal form, if it is an application of a built-in function to too few arguments, an application of a constructor, or a lambda abstraction. Moreover, if a value is not of the form **error** v, it is called *defined*:

$$\begin{array}{lll} v ::= c\, e_1 \dots\ e_n & n < \mathsf{ar}(f) & \\ \quad |F\ e_1 \dots\ e_n & n = \mathsf{ar}(F), \forall i \in \mathsf{strict}(F) & e_i \text{ is defined value} \\ \quad |\lambda x \to e & & \end{array}$$

An even more restricted subset of the set of values is the set of *constructor values* which are expressions in *constructor head normal form.* It is similar to weak head normal form, but with the additional restriction, that arguments of a fully applied constructor are in constructor normal form as well:

$$\begin{array}{lll} V ::= c\, e_1 \dots\ e_n & n < \mathsf{ar}(f) & \\ \quad |F\ V_1 \dots\ V_n & n = \mathsf{ar}(F), \forall i \in \mathsf{strict}(F) & V_i \text{ is defined} \\ \quad |\lambda x \to e & & \end{array}$$

To further simplify the presentation we introduce evaluation contexts. The following evaluation context $\mathbb{E}$ corresponds to weak head normal forms:

$$\begin{array}{ll} \mathbb{E} ::= [\cdot] \mid \mathbb{E}\ e \mid \textbf{type}\ x = \mathbb{E}\ \textbf{of}\ \{r \to e_1; \_ \to e_2\} & \\ \quad |c\ e_1\ \dots e_{i-1}\ \mathbb{E}\ e_{i+1}\ \dots\ e_n & i \in \mathsf{strict}(c), n = \mathsf{ar}(c), \\ & \forall j < i, j \in \mathsf{strict}(c)\colon e_j \text{ is defined value} \end{array}$$

The evaluation context $\mathbb{F}$ corresponds to constructor head normal forms:

$$\begin{array}{ll} \mathbb{F} ::= [\cdot] \mid \mathbb{E}\ e \mid \textbf{type}\ x = \mathbb{E}\ \textbf{of}\ \{r \to e_1; \_ \to e_2\} & \\ \quad |f\ e_1\ \dots e_{i-1}\ \mathbb{E}\ e_{i+1}\ \dots\ e_n & i \in \mathsf{strict}(f), n = \mathsf{ar}(f), \\ & \forall j < i, j \in \mathsf{strict}(f)\colon e_j \text{ is defined value} \\ \quad |F\ V_1\ \dots V_{i-1}\ \mathbb{F}\ e_{i+1}\ \dots\ e_n & n = \mathsf{ar}(F), V_1\ \dots V_{i-1} \text{ are defined} \end{array}$$

Computations take place in a context of an *event log*, i.e. a sequence of values of type Event. In the following definition of the semantics of the report language we use $(ev_i)_{i<n}$ to refer to this sequence, where each ev_i is of the form $r\{\overline{f_j = e_j}\}$ with $r \leq \mathsf{Event}$.

We assume that the Event record type has a field internalTimeStamp that records the time at which the event was added to the log. For each ev_i, we define its extension ev'_i as follows: Each occurrence of an entity value $\langle r, e\rangle$ is replaced by $\langle r, e, t\rangle$ where t is the value of the internalTimeStamp field of ev_i. This will allow us to define the semantics of the contextual dereference operator

@. The semantics of both the @ and the ! operator are given by the lookup operator, which is provided by the entity store, compare Section 2.3. In order to retrieve the latest value associated to an entity, we assume the timestamp t_{now} that denotes the current time.

The rules describing the semantics of the report language in the form of a small step transition relation $\rightarrow$ are given in Figure 14.

$$\frac{e \to e'}{\mathbb{F}\,[e] \to \mathbb{F}\,[e']}\ \text{(Context)} \qquad \frac{}{\mathbb{F}\,[\textbf{error}\ v] \to \textbf{error}\ v}\ \text{(Error)}$$

$$\frac{}{(\lambda x \to e_1)e_2 \to e_1\,[x/e_2]}\ \text{(Abs)} \qquad \frac{}{\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \to e_2\,[x/e_1]}\ \text{(Let)}$$

$$\frac{r' \leq r \qquad v = r'\{\dots\}}{\textbf{type}\ x = v\ \textbf{of}\ \{r \to e_1;\, \_ \to e_2\} \to e_1\,[x/v]}\ \text{(Type suc)}$$

$$\frac{r' \not\leq r \qquad v = r'\{\dots\}}{\textbf{type}\ x = v\ \textbf{of}\ \{r \to e_1;\, \_ \to e_2\} \to e_2\,[x/v]}\ \text{(Type def)}$$

$$\frac{\begin{array}{l}\text{injection } \phi\colon \{1,\dots,m\} \hookrightarrow \{1,\dots,n\} \qquad e''_i = \begin{cases} e'_{\phi^{-1}(i)} & \text{if } i \in \mathsf{Im}(\phi) \\ e_i & \text{otherwise}\end{cases} \\ \forall j \in \{1,\dots,m\}:\ f'_j = f_{\phi(j)}\end{array}}{r\{f_1 = e_1,\dots,f_n = e_n\}\ \{f'_1 = e'_1,\dots,f'_m = e'_m\} \to r\{f_1 = e''_1,\dots,f_n = e''_n\}}\ \text{(Mod)}$$

$$\frac{}{r\{f_1 = e_1,\dots,f_n = e_n\}.f_i \to e_i}\ \text{(Acc)} \qquad \frac{}{\textbf{if True then}\ e_1\ \textbf{else}\ e_2 \to e_1}\ \text{(If True)}$$

$$\frac{}{\textbf{if False then}\ e_1\ \textbf{else}\ e_2 \to e_2}\ \text{(If False)} \qquad \frac{}{\textbf{case}\ (\textbf{Inl}\ e)\ e_1\ e_2 \to e_1\ e}\ \text{(Case Left)}$$

$$\frac{}{\textbf{case}\ (\textbf{Inr}\ e)\ e_1\ e_2 \to e_2\ e}\ \text{(Case Right)} \qquad \frac{i \in \{1,2\}}{(e_1,e_2).i \to e_i}\ \text{(Proj)}$$

$$\frac{}{\textbf{events} \to [ev_1, ev_2, \dots, ev_n]}\ \text{(Events)} \qquad \frac{}{\textbf{fold}\ e_1\ e_2\ \ [] \to e_2}\ \text{(Fold Empty)}$$

$$\frac{}{\textbf{fold}\ e_1\ e_2\ (e_3 \# e_4) \to e_1\ e_3\ (\textbf{fold}\ e_1\ e_2\ e_4)}\ \text{(Fold Cons)}$$

$$\frac{\text{lookup}_{t_{\text{now}}}(e, t_{\text{now}}) = v}{\langle r, e, t\rangle ! \to v}\ \text{(! ignore)} \qquad \frac{\text{lookup}_{t_{\text{now}}}(e, t_{\text{now}}) = v}{\langle r, e\rangle ! \to v}\ \text{(!)}$$

$$\frac{\text{lookup}_{t_{\text{now}}}(e, t) = v}{\langle r, e, t\rangle\,@ \to v}\ \text{(@)} \qquad \frac{\text{lookup}_{t_{\text{now}}}(e, t_{\text{now}}) = v}{\langle r, e\rangle\,@ \to v}\ \text{(@ now)}$$

Figure 14: Small step operational semantics of the report language.

C μERP Specification

C.1 Ontology

C.1.1 Data

ResourceType is Data.
ResourceType is abstract.

Currency is a ResourceType.
Currency is abstract.

DKK is a Currency.
EUR is a Currency.

ItemType is a ResourceType.
ItemType is abstract.

Bicycle is an ItemType.
Bicycle has a String called model.

Resource is Data.
Resource is abstract.

Money is a Resource.
Money has a Currency.
Money has a Real called amount.

Item is a Resource.
Item has an ItemType.
Item has a Real called quantity.

Agent is Data.

Me is an Agent.

Customer is an Agent.
Customer has a String called name.
Customer has an Address.

Vendor is an Agent.
Vendor has a String called name.
Vendor has an Address.

Address is Data.
Address has a String.
Address has a Country.

Country is Data.
Country is abstract.

Denmark is a Country.

OrderLine is Data.
OrderLine has an Item.
OrderLine has Money called unitPrice.
OrderLine has a Real called vatPercentage.

CurrentAssets is Data.
CurrentAssets has a list of Money called currentAssets.
CurrentAssets has a list of Money called inventory.
CurrentAssets has a list of Money called accountsReceivable.
CurrentAssets has a list of Money called cashPlusEquiv.

Liabilities is Data.
Liabilities has a list of Money called liabilities.

Liabilities has a list of Money called accountsPayable.
Liabilities has a list of Money called vatPayable.

Invoice is Data.
Invoice has an Agent called sender.
Invoice has an Agent called receiver.
Invoice has a list of OrderLine called orderLines.

UnpaidInvoice is Data.
UnpaidInvoice has an Invoice.
UnpaidInvoice has a list of Money called remainder.

CustomerStatistics is Data.
CustomerStatistics has a Customer entity.
CustomerStatistics has Money called totalPaid.

C.1.2 Transaction

BiTransaction is a Transaction.
BiTransaction is abstract.
BiTransaction has an Agent entity called sender.
BiTransaction has an Agent entity called receiver.

Transfer is a BiTransaction.
Transfer is abstract.

Payment is a Transfer.
Payment is abstract.
Payment has Money.

CashPayment is a Payment.
CreditCardPayment is a Payment.
BankTransfer is a Payment.

Delivery is a Transfer.
Delivery has a list of Item called items.

IssueInvoice is a BiTransaction.
IssueInvoice has a list of OrderLine called orderLines.

RequestRepair is a BiTransaction.
RequestRepair has a list of Item called items.

Repair is a BiTransaction.
Repair has a list of Item called items.

C.1.3 Report

IncomeStatement is a Report.
IncomeStatement has a list of Money called revenue.
IncomeStatement has a list of Money called costOfGoodsSold.
IncomeStatement has a list of Money called contribMargin.
IncomeStatement has a list of Money called fixedCosts.
IncomeStatement has a list of Money called depreciation.
IncomeStatement has a list of Money called netOpIncome.

BalanceSheet is a Report.
BalanceSheet has a list of Money called fixedAssets.
BalanceSheet has CurrentAssets.
BalanceSheet has a list of Money called totalAssets.
BalanceSheet has Liabilities.
BalanceSheet has a list of Money called ownersEquity.
BalanceSheet has a list of Money called totalLiabilitiesPlusEquity.

CashFlowStatement is a Report.
CashFlowStatement has a list of Payment called expenses.
CashFlowStatement has a list of Payment called revenues.

CashFlowStatement has a list of Money called revenueTotal.
CashFlowStatement has a list of Money called expenseTotal.

UnpaidInvoices is a Report.
UnpaidInvoices has a list of UnpaidInvoice called invoices.

VATReport is a Report.
VATReport has a list of Money called outgoingVAT.
VATReport has a list of Money called incomingVAT.
VATReport has a list of Money called vatDue.

Inventory is a Report.
Inventory has a list of Item called availableItems.

TopNCustomers is a Report.
TopNCustomers has a list of CustomerStatistics.

C.1.4 Contract

Purchase is a Contract.
Purchase has a Vendor entity.
Purchase has a list of OrderLine called orderLines.

Sale is a Contract.
Sale has a Customer entity.
Sale has a list of OrderLine called orderLines.

C.2 Reports

C.2.1 Prelude Functions

```
-- Arithmetic
min : (Ord a) ⇒ a → a → a
min x y = if x < y then x else y

max : (Ord a) ⇒ a → a → a
max x y = if x > y then x else y

-- List functions
null : [a] → Bool
null = fold (λe r → False) True

first : a → [a] → a
first = fold (λx a → x)

head : [a] → a
head = first (error "'head' applied to empty list")

elemBy : (a → a → Bool) → a → [a] → Bool
elemBy f e = fold (λx a → a ∨ f x e) False

elem : (Ord a) ⇒ a → [a] → Bool
elem = elemBy (≡)

sum : (a < Real, Int < a) ⇒ [a] → a
sum = fold (+) 0

length : [a] → Int
length = fold (λ x y → y+1) 0

map : (a → b) → [a] → [b]
map f = fold (λx a → (f x) # a) []

filter : (a → Bool) → [a] → [a]
filter f = fold (λx a → if f x then x # a else a) []
```

```
nupBy : (a → a → Bool) → [a] → [a]
nupBy f = fold (λx a → x # filter (λ y → ¬ (f x y)) a) []

nup : (Ord a) ⇒ [a] → [a]
nup = nupBy (≡)

all : (a → Bool) → [a] → Bool
all f = fold (λx a → f x ∧ a) True

any : (a → Bool) → [a] → Bool
any f = fold (λx a → f x ∨ a) False

concat : [[a]] → [a]
concat = fold (λx a → x ++ a) []

concatMap : (a → [b]) → [a] → [b]
concatMap f l = concat (map f l)

take : Int → [a] → [a]
take n l = (fold (λx a → if a.2 > 0 then (x # a.1,a.2 − 1) else a) ([],n) l).1

−− Grouping functions
addGroupBy : (a → a → Bool) → a → [[a]] → [[a]]
addGroupBy f a ll =
  let felem l = fold (λ el r → f el a) False l
      run el r =
        if r.1 then (True,el # r.2)
        else if felem el then (True, (a # el) # r.2)
        else (False, el # r.2)
      res = fold run (False,[]) ll
  in if res.1 then res.2 else [a] # res.2

groupBy : (a → a → Bool) → [a] → [[a]]
groupBy f = fold (addGroupBy f) []

addGroupProj : (Ord b) ⇒ (a → b) → a → [(b,[a])] → [(b,[a])]
addGroupProj f a ll =
  let run el r =
        if r.1 then(True,el # r.2)
        else if el.1 ≡ f a then (True, (el.1,a # el.2) # r.2)
        else (False, el # r.2)
      res = fold run (False,[]) ll
  in if res.1 then res.2 else  (f a,[a]) # res.2

groupProj : (Ord b) ⇒ (a → b) → [a] → [(b, [a])]
groupProj f = fold (addGroupProj f) []


−− Sorting functions
insertBy : (a → a → Bool) → a → [a] → [a]
insertBy le a l =
  let ins e r =
        if r.1 then (True, e # r.2)
        else if le e a then (True,e # a # r.2)
        else (False, e # r.2)
      res = fold ins (False,[]) l
  in if res.1 then res.2 else a # res.2

insertProj : (Ord b) ⇒ (a → b) → a → [a] → [a]
insertProj proj = insertBy (λx y → proj x ≤ proj y)

insert : (Ord a) ⇒ a → [a] → [a]
insert = insertBy (≤)

sortBy : (a → a → Bool) → [a] → [a]
sortBy le = fold (λe r → insertBy le e r) []

sortProj : (Ord b) ⇒ (a → b) → [a] → [a]
sortProj proj = sortBy (λx y → proj x ≤ proj y)

sort : (Ord a) ⇒ [a] → [a]
sort = sortBy (≤)


−− Generators for 'lifecycled' data
reports : [PutReport]
reports = nupBy (λpr1 pr2 → pr1.name ≡ pr2.name) [pr |
  cr : CreateReport ← events,
  pr : PutReport = first cr [ur | ur : ReportEvent ← events, ur.name ≡ cr.name]]

entities : [(⟨Data⟩,String)]
entities = [(ce.ent,ce.recordType) |
  ce : CreateEntity ← events,
  null [de | de : DeleteEntity ← events, de.ent ≡ ce.ent]]

contracts : [PutContract]
contracts = [pc |
  cc : CreateContract ← events,
  pc = first cc [uc | uc : UpdateContract ← events, uc.contractId ≡ cc.contractId],
  null [cc | cc : ConcludeContract ← events, cc.contractId ≡ pc.contractId]]

contractDefs : [PutContractDef]
contractDefs = nupBy (λpcd1 pcd2 → pcd1.name ≡ pcd2.name) [pcd |
  ccd : CreateContractDef ← events,
  pcd : PutContractDef = first ccd [ucd | ucd : ContractDefEvent ← events, ucd.name ≡ ccd.name]]

transactionEvents : [TransactionEvent]
transactionEvents = [tr | tr : TransactionEvent ← events]

transactions : [Transaction]
transactions = [tr.transaction | tr ← transactionEvents]
```

C.2.2 Domain-Specific Prelude Functions

```
−− Check if an agent is the company itself
isMe : ⟨Agent⟩ → Bool
isMe a = a :? ⟨Me⟩

−− Normalise a list of money by grouping currencies together
normaliseMoney : [Money] → [Money]
normaliseMoney ms = [Money{currency = m.1, amount = sum (map (λm → m.amount) m.2)} |
  m ← groupProj (λm → m.currency) ms]

−− Add one list of money from another
addMoney : [Money] → [Money] → [Money]
addMoney m1 m2 = normaliseMoney (m1 ++ m2)

−− Subtract one list of money from another
subtractMoney : [Money] → [Money] → [Money]
subtractMoney m1 m2 = addMoney m1 (map (λm → m{amount = 0 − m.amount}) m2)

−− Produce normalised list of all items given in list
normaliseItems : [Item] → [Item]
normaliseItems its = [Item{itemType = i.1, quantity = sum (map (λis → is.quantity) i.2)} |
  i ← groupProj (λis → is.itemType) its]

−− List of all invoices and their associated contract ID
invoices : [(Int,IssueInvoice)]
invoices = [(tr.contractId,inv) |
  tr ← transactionEvents,
  inv : IssueInvoice = tr.transaction]

−− List of all received invoices and their associated contract ID
invoicesReceived : [(Int,IssueInvoice)]
invoicesReceived =
  filter (λinv → ¬ (isMe (inv.2).sender) ∧ isMe (inv.2).receiver) invoices
```

```
-- List of all sent invoices and their associated contract ID
invoicesSent : [(Int,IssueInvoice)]
invoicesSent = filter (λinv → isMe inv.2.sender ∧ ¬ (isMe inv.2.receiver)) invoices

-- Calculate the total price including VAT on an invoice
invoiceTotal : (a.orderLines : [OrderLine]) ⇒ a → [Money]
invoiceTotal inv = normaliseMoney [line.unitPrice{amount = price} |
  line ← inv.orderLines,
  quantity = line.item.quantity,
  price = ((100 + line.vatPercentage) × line.unitPrice.amount × quantity) / 100]

-- List of all items delivered to the company
itemsReceived : [Item]
itemsReceived = normaliseItems [is |
  tr ← transactionEvents,
  del : Delivery = tr.transaction,
  ¬(isMe del.sender) ∧ isMe del.receiver,
  is ← del.items]

-- List of all items that have been sold
itemsSold : [Item]
itemsSold = normaliseItems [line.item | inv ← invoicesSent, line ← inv.2.orderLines]

-- Inventory acquisitions, that is a list of all received items and the unit
-- price of each item, exluding VAT.
invAcq : [(Item,Money)]
invAcq = [(item,line.unitPrice) |
  inv ← invoicesReceived,
  tr ← transactionEvents,
  tr.contractId ≡ inv.1,
  deliv : Delivery = tr.transaction,
  item ← deliv.items,
  line ← inv.2.orderLines,
  line.item.itemType ≡ item.itemType]

-- FIFO costing: Calculate the cost of all sold goods based on FIFO costing.
fifoCost : [Money]
fifoCost = let
   -- Check whether a set of items equals the current set of items in the
   -- inventory. If so, 'take' as many of the inventory items as possible
   -- and add the price of these items to the totals.
   checkInventory y x = let
       invItem = y.1  -- The current item in the inventory
       invPrice = y.2 -- The price of the current item in the inventory
       oldInv = x.1 -- The part of the inventory that has been processed
       item = x.2 -- The item to find in the inventory
       total = x.3 -- The total costs so far
     in
     if item.itemType ≡ invItem.itemType then let
         deltaInv =
           if invItem.quantity ≤ item.quantity then
             []
           else
             [(invItem{quantity = invItem.quantity − item.quantity},invPrice)]
         remainingItem = item{quantity = max 0 (item.quantity − invItem.quantity)}
         price = invPrice{amount = invPrice.amount × (min item.quantity invItem.quantity)}
       in
       (oldInv ++ deltaInv, remainingItem, price # total)
     else
       (oldInv ++ [(invItem,invPrice)], item, total)

   -- Process a sold item
   processSoldItem soldItem x = let
       total = x.1 -- the total costs so far
       inv = x.2   -- the remaning inventory so far
       y = fold checkInventory ([],soldItem,total) inv
     in
```

```
    (y.3,y.1)
  in
  normaliseMoney ((fold processSoldItem ([],invAcq) itemsSold).1)

-- Outoing VAT
vatOutgoing : [Money]
vatOutgoing = normaliseMoney [price |
  inv ← invoicesReceived,
  l ← inv.2.orderLines,
  price = l.unitPrice{amount = (l.vatPercentage × l.unitPrice.amount × l.item.quantity) / 100}]

-- Incoming VAT
vatIncoming : [Money]
vatIncoming = normaliseMoney [price |
  inv ← invoicesSent,
  l ← inv.2.orderLines,
  price = l.unitPrice{amount = (l.vatPercentage × l.unitPrice.amount × l.item.quantity) / 100}]
```

C.2.3 Internal Reports

Me

name: *Me*
description:
Returns the pseudo entity 'Me' that represents the company.
tags: *internal*, *entity*

```
report : ⟨Me⟩
report = head [me | me : ⟨Me⟩ ← map (λe → e.1) entities]
```

Entities

name: *Entities*
description:
A list of all entities.
tags: *internal*, *entity*

```
report : [⟨Data⟩]
report = map (λe → e.1) entities
```

EntitiesByType

name: *EntitiesByType*
description:
A list of all entities with the given type.
tags: *internal*, *entity*

```
report : String → [⟨Data⟩]
report t = map (λe → e.1) (filter (λe → e.2 ≡ t) entities)
```

ReportNames

name: *ReportNames*
description:
A list of names of all registered reports.
tags: *internal*, *report*

```
report : [String]
report = [r.name | r ← reports]
```

ReportNamesByTags

name: *ReportNamesByTags*
description:
A list of reports that have the all **tags** *provided as first argument to the function and none of the* **tags** *provided as second argument.*
tags: *internal*, *report*

```
filt allOf noneOf rep =
  all (λx → elem x rep.tags) allOf ∧
  ¬ (any (λx → elem x rep.tags) noneOf)

report : [String] → [String] → [String]
report allOf noneOf = [r.name | r ← filter (filt allOf noneOf) reports]
```

ReportTags

name: *ReportTags*
description:
A list of **tags** *that are used in registered reports.*
tags: *internal*, *report*

```
report : [String]
report = nup (concatMap (λx → x.tags) reports)
```

ContractTemplates

name: *ContractTemplates*
description:
A list of 'PutContractDef' events for each non−deleted contract template.
tags: *internal*, *contract*

```
report : [PutContractDef]
report = contractDefs
```

ContractTemplatesByType

name: *ContractTemplatesByType*
description:
A list of 'PutContractDef' events for each non−deleted contract template of the given type.
tags: *internal*, *contract*

```
report : String → [PutContractDef]
report r = filter (λx → x.recordType ≡ r) contractDefs
```

Contracts

name: *Contracts*
description:
A list of all running (i.e. non−concluded) contracts.
tags: *internal*, *contract*

```
report : [PutContract]
report = contracts
```

ContractHistory

name: *ContractHistory*
description:
A list of previous transactions for the given contract.
tags: *internal*, *contract*

```
report : Int → [TransactionEvent]
report cid = [transaction |
  transaction : TransactionEvent ← events,
  transaction.contractId ≡ cid]
```

ContractSummary

name: *ContractSummary*
description:
A list of meta data for the given contract.
tags: *internal*, *contract*

```
report : Int → [PutContract]
report cid = [createCon |
  createCon : PutContract ← contracts,
  createCon.contractId ≡ cid]
```

C.2.4 External Reports

IncomeStatement

name: *IncomeStatement*
description:
The Income Statement.
tags: *external*, *financial*

```
-- Revenue
revenue = normaliseMoney [line.unitPrice{amount = amount} |
 inv ← invoicesSent,
 line ← inv.2.orderLines,
 amount = line.unitPrice.amount × line.items.numberOfItems]

costOfGoodsSold = fifoCost
contribMargin = subtractMoney revenue fifoCost
fixedCosts = [] -- For simplicity
depreciation = [] -- For simplicity
netOpIncome = subtractMoney (subtractMoney contribMargin fixedCosts) depreciation

report : IncomeStatement
report = IncomeStatement{
  revenue = revenue,
  costOfGoodsSold = costOfGoodsSold,
  contribMargin = contribMargin,
  fixedCosts = fixedCosts,
  depreciation = depreciation,
  netOpIncome = netOpIncome}
```

BalanceSheet

name: *BalanceSheet*
description:
The Balance Sheet.
tags: *external*, *financial*

```
-- List of all payments and their associated contract ID
payments : [(Int,Payment)]
payments = [ (tr.contractId,payment) |
  tr ← transactionEvents,
  payment : Payment = tr.transaction]

-- List of all received payments and their associated contract ID
paymentsReceived : [(Int,Payment)]
paymentsReceived = filter (λp → ¬ (isMe p.2.sender) ∧ isMe p.2.receiver) payments

-- List of all payments made and their associated contract ID
paymentsMade : [(Int,Payment)]
paymentsMade = filter (λp → isMe p.2.sender ∧ ¬ (isMe p.2.receiver)) payments

cashReceived : [Money]
cashReceived = normaliseMoney (map (λp → p.2.money) paymentsReceived)

cashPaid : [Money]
cashPaid = normaliseMoney (map (λp → p.2.money) paymentsMade)

netCashFlow : [Money]
netCashFlow = subtractMoney cashReceived cashPaid

depreciation : [Money]
depreciation = [] -- For simplicity

fAssetAcq : [Money]
fAssetAcq = [] -- For simplicity

fixedAssets : [Money]
fixedAssets = subtractMoney fAssetAcq depreciation

inventory : [Money]
inventory =
  let inventoryValue = [price |
        item ← invAcq,
        price = item.2{amount = item.2.amount × item.1.quantity}]
  in
  subtractMoney inventoryValue fifoCost

accReceivable : [Money]
accReceivable =
  let paymentsDue = normaliseMoney [line.unitPrice{amount = amount} |
        inv ← invoicesSent,
        line ← inv.2.orderLines,
        amount = line.unitPrice.amount × line.item.quantity]
  in
  subtractMoney paymentsDue cashReceived

currentAssets : [Money]
currentAssets = addMoney inventory (addMoney accReceivable netCashFlow)

totalAssets : [Money]
totalAssets = addMoney fixedAssets currentAssets

accPayable : [Money]
accPayable =
  let paymentsDue = [line.unitPrice{amount = amount} |
        inv ← invoicesReceived,
        line ← inv.2.orderLines,
        amount = line.unitPrice.amount × line.item.quantity]
  in
  subtractMoney paymentsDue cashPaid

vatPayable : [Money]
vatPayable = subtractMoney vatIncoming vatOutgoing

liabilities : [Money]
liabilities = addMoney accPayable vatPayable

ownersEq : [Money]
ownersEq = subtractMoney totalAssets liabilities

totalLiabPlusEq : [Money]
totalLiabPlusEq = addMoney liabilities ownersEq

report : BalanceSheet
report = BalanceSheet{
  fixedAssets = fixedAssets,
  currentAssets = CurrentAssets{
    currentAssets = currentAssets,
    inventory = inventory,
    accountsReceivable = accReceivable,
    cashPlusEquiv = netCashFlow},
  totalAssets = totalAssets,
  liabilities = Liabilities{
    liabilities = liabilities,
    accountsPayable = accPayable,
    vatPayable = vatPayable},
  ownersEquity = ownersEq,
  totalLiabilitiesPlusEquity = totalLiabPlusEq}
```

CashFlowStatement

name: *CashFlowStatement*
description:
  The Cash Flow Statement.
tags: *external*, *financial*

```
sumPayments : [Payment] → [Money]
sumPayments ps = normaliseMoney (map (λp → p.money) ps)

report : CashFlowStatement
report = let
    payments = [payment | payment : Payment ← transactions]
    mRevenues = [payment | payment ← payments, isMe (payment.receiver)]
    mExpenses = [payment | payment ← payments, isMe (payment.sender)]
  in
  CashFlowStatement{
    revenues = mRevenues,
    expenses = mExpenses,
    revenueTotal = sumPayments mRevenues,
    expenseTotal = sumPayments mExpenses}
```

UnpaidInvoices

name: *UnpaidInvoices*
description:
  A list of unpaid invoices.
tags: *external*, *financial*

```
-- Generate a list of unpaid invoices
unpaidInvoices : [UnpaidInvoice]
unpaidInvoices = [UnpaidInvoice{invoice = inv, remainder = remainder} |
  invS ← invoicesSent,
  inv = Invoice{
    sender = invS.2.sender @,
    receiver = invS.2.receiver @,
    orderLines = invS.2.orderLines},
  payments = [payment.money |
    tr ← transactionEvents,
    tr.contractId ≡ invS.1,
    payment : Payment = tr.transaction],
  remainder = subtractMoney (invoiceTotal inv) payments,
  any (λm → m.amount > 0) remainder]

report : UnpaidInvoices
report = UnpaidInvoices{invoices = unpaidInvoices}
```

VATReport

name: *VATReport*
description:
The VAT report.
tags: *external*, *financial*

```
report : VATReport
report = VATReport{
  outgoingVAT = vatOutgoing,
  incomingVAT = vatIncoming,
  vatDue = subtractMoney vatIncoming vatOutgoing}
```

Inventory

name: *Inventory*
description:
A list of items in the inventory available for sale (regardless of whether we have paid for them).
tags: *external*, *inventory*

```
report : Inventory
report =
  let itemsSold' = map (λi → i{quantity = 0 − i.quantity}) itemsSold
  in
  −− The available items is the list of received items minus the list of reserved
  −− or sold items
  Inventory{availableItems = normaliseItems (itemsReceived ++ itemsSold')}
```

TopNCustomers

name: *TopNCustomers*
description:
A list of customers who have spent must money in the given currency.
tags: *external*, *financial*, *crm*

```
customers : [⟨Customer⟩]
customers = [c | c : ⟨Customer⟩ ← map (λe → e.1) entities]

totalPayments : Currency → ⟨Customer⟩ → Real
totalPayments c cu = sum [d |
  p : Payment ← transactions,
  p.sender ≡ cu ∨ p.receiver ≡ cu,
  p.money.currency ≡ c,
  d = if p.sender ≡ cu then p.money.amount else 0 − p.money.amount]

customerStatistics : Currency → [CustomerStatistics]
customerStatistics c = [CustomerStatistics{customer = cu, totalPaid = p} |
  cu ← customers,
  p = Money{currency = c, amount = totalPayments c cu}]

topN : Int → [CustomerStatistics] → [CustomerStatistics]
topN n cs = take n (sortBy (λcs1 cs2 → cs1.totalPaid > cs2.totalPaid) cs)

report : Int → Currency → TopNCustomers
report n c = TopNCustomers{customerStatistics = topN n (customerStatistics c)}
```

C.3 Contracts

C.3.1 Prelude

```
// Arithmetic
fun floor x = let n = ceil x in if n > x then n − 1 else n
fun round x = let n1 = ceil x in let n2 = floor x in if n1 + n2 > 2 × x then n2 else n1
fun max a b = if a > b then a else b
fun min a b = if a > b then b else a

// List functions
fun filter f = foldr (λx b → if f x then x # b else b) []
fun map f = foldr (λx b → (f x) # b) []
val length = foldr (λx b → b + 1) 0
fun null l = l ≡ []
fun elem x = foldr (λy b → x ≡ y ∨ b) false
fun all f = foldr (λx b → b ∧ f x) true
fun any f = foldr (λx b → b ∨ f x) false
val reverse = foldl (λa e → e # a) []
fun append l1 l2 = foldr (λe a → e # a) l2 l1

// Lists as sets
fun subset l1 l2 = all (λx → elem x l2) l1
fun diff l1 l2 = filter (λx → ¬ (elem x l2)) l1
```

C.3.2 Domain-Specific Prelude

```
// Check if 'lines' are in stock by invoking the 'Inventory' report
fun inStock lines =
  let inv = (reports.inventory ()).availableItems
  in
  all (λl → any (λi → (l.item).itemType ≡ i.itemType ∧ (l.item).quantity ≤ i.quantity) inv) lines

// Check that amount 'm' equals the total amount in m's currency of a list of sales lines
fun checkAmount m orderLines =
  let a = foldr (λx acc →
            if (x.unitPrice).currency ≡ m.currency then
              (x.item).quantity × (100 + x.vatPercentage) × (x.unitPrice).amount + acc
            else
              acc) 0 orderLines
  in
  m.amount × 100 ≡ a

// Remove sales lines that have the currency of 'm'
fun remainingOrderLines m = filter (λx → (x.unitPrice).currency ≢ m.currency)

// A reference to the designated entity that represents the company
val me = reports.me ()
```

C.3.3 Contract Templates

Purchase

```
name: purchase
type: Purchase
description: "Set up a purchase"

clause purchase(lines : [OrderLine])⟨me : ⟨Me⟩, vendor : ⟨Vendor⟩⟩ =
 ⟨vendor⟩ Delivery(sender s, receiver r, items i)
   where s ≡ vendor ∧ r ≡ me ∧ i ≡ map (λx → x.item) lines
   due within 1W
 then
 when IssueInvoice(sender s, receiver r, orderLines sl)
   where s ≡ vendor ∧ r ≡ me ∧ sl ≡ lines
   due within 1Y
 then
 payment(lines, vendor, 14D)⟨me⟩

clause payment(lines : [OrderLine], vendor : ⟨Vendor⟩, deadline : Duration)
                    ⟨me : ⟨Me⟩⟩ =
 if null lines then
   fulfilment
 else
   ⟨me⟩ BankTransfer(sender s, receiver r, money m)
     where s ≡ me ∧ r ≡ vendor ∧ checkAmount m lines
     due within deadline
     remaining newDeadline
   then
   payment(remainingOrderLines m lines, vendor, newDeadline)⟨me⟩

contract = purchase(orderLines)⟨me, vendor⟩
```

Sale

name: *sale*
type: Sale
description: `"Set up a sale"`

```
clause sale(lines : [OrderLine])⟨me : ⟨Me⟩, customer : ⟨Customer⟩⟩ =
 ⟨me⟩ IssueInvoice(sender s, receiver r, orderLines sl)
   where s ≡ me ∧ r ≡ customer ∧ sl ≡ lines ∧ inStock lines
   due within 1H
 then
 payment(lines, me, 10m)⟨customer⟩
 and
 ⟨me⟩ Delivery(sender s, receiver r, items i)
   where s ≡ me ∧ r ≡ customer ∧ i ≡ map (λx → x.item) lines
   due within 1W
 then
 repair(map (λx → x.item) lines, customer, 3M)⟨me⟩

clause payment(lines : [OrderLine], me : ⟨Me⟩, deadline : Duration)
                    ⟨customer : ⟨Customer⟩⟩ =
 if null lines then
   fulfilment
 else
   ⟨customer⟩ Payment(sender s, receiver r, money m)
     where s ≡ customer ∧ r ≡ me ∧ checkAmount m lines
     due within deadline
     remaining newDeadline
   then
   payment(remainingOrderLines m lines, me, newDeadline)⟨customer⟩

clause repair(items : [Item], customer : ⟨Customer⟩, deadline : Duration)
                  ⟨me : ⟨Me⟩⟩ =
 when RequestRepair(sender s, receiver r, items i)
   where s ≡ customer ∧ r ≡ me ∧ subset i items
   due within deadline
   remaining newDeadline
 then
 ⟨me⟩ Repair(sender s, receiver r, items i')
   where s ≡ me ∧ r ≡ customer ∧ i ≡ i'
   due within 5D
 and
 repair(items, customer, newDeadline)⟨me⟩

contract = sale(orderLines)⟨me, customer⟩
```