Journal of Logical and Algebraic Methods in Programming 121 (2021) 100634

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Semantics of multiway dataflow constraint systems

Magne Haveraaen[a], Jaakko Järvi[b,*]

[a] University of Bergen, Norway
[b] University of Turku, Finland

ARTICLE INFO

Article history:
Received 27 March 2019
Received in revised form 29 November 2020
Accepted 30 November 2020
Available online 3 December 2020

Keywords:
Dataflow constraint systems
Institutions
Reuse
Graphical user interfaces
Module system
Verification and testing

ABSTRACT

Multiway dataflow constraint systems (MDCS) is a programming model where statements are not executed in a predetermined order. Rather, individual methods are selected from specific method sets and then executed to achieve a desired global state. The selection is done by a *planner*, which typically bases the choice of methods on the history of updates to the global state. MDCS is well suited for describing user interface logic where choosing what code to execute depends in complicated ways on the history of user interactions and on data availability. User interfaces are the domain of examples in this paper.

Much of the research into MDCS has been on planning algorithms and their efficiency. Here we investigate a semantic setting for MDCS, introducing *dataflow constraints* as modules with explicit goals and related method sets. MDCS is defined in a similar manner, with an explicit goal and a set of supporting dataflow constraints. This enables verification and testing of methods and dataflow constraints against the goals. The exposition is based on abstract syntax for an idealised programming language with global variables. On top of this we define a modular reuse mechanism for dataflow constraints based on Goguen-Burstall *institution theory*. We show how this setup enables reuse in user interfaces; traditionally code that defines user interface logic is almost invariably non-reusable.

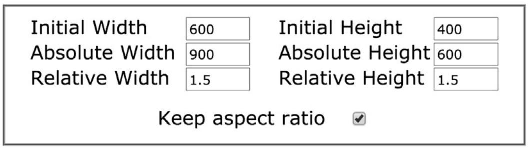
1. Introduction

A *dataflow constraint* describes a relation amongst variables and means to satisfy that relation. The latter is a set of functions, *constraint satisfaction methods*, that compute values for some of the relation's variables, using others as input. A collection of dataflow constraints to be satisfied simultaneously is a *(dataflow) constraint system*. There are many variations of dataflow constraints and constraint solving algorithms [1–4]. The one we focus on here is *multiway dataflow constraint systems* (MDCS), where information can flow in different directions between a set of variables. Our prior work on dataflow constraint solving include mechanisms for specialising fast solver programs when the constraint system is known statically [5] and a study of the properties of composing dataflows [6].

Dataflow constraints can increase the abstraction level of event-based programming: when programming with dataflow constraints, the programmer declares a dependency between variables but leaves the details of maintaining that dependency, what computations and assignments need to be performed and when, to the constraint system. The basic mode of operation is that whenever the value of a variable in a constraint system is modified (triggered by an event from outside of the system), the constraint solver computes a new variable valuation that satisfies all constraints in the system.

[*] Corresponding author.
E-mail addresses: magne.haveraaen@ii.uib.no (M. Haveraaen), jaakko.jarvi@utu.fi (J. Järvi).

https://doi.org/10.1016/j.jlamp.2020.100634
2352-2208/



Initial Width 600 Initial Height 400
Absolute Width 900 Absolute Height 600
Relative Width 1.5 Relative Height 1.5

Keep aspect ratio ☑

Fig. 1. An example GUI implemented using a constraint system.

Applications of dataflow constraint systems include frameworks for graphical user interface (GUI) programming, where constraint systems are used for tasks such as layout and enforcing dependencies amongst user-modifiable values [7–9] (several contemporary JavaScript GUI libraries, such as *Angular*, *Knockout*, and *Backbone*, support simple forms of dataflow constraints); tools for collaborative work [10]; and spreadsheet applications. Dataflow constraint systems have also been integrated to general purpose host languages to be applied for varied programming tasks [11,12].

This paper investigates the semantical underpinnings of constraint systems. Our goal is to provide a formal framework for reasoning about constraint systems and their compositions. The main contributions of this paper are based on a perspective on constraint systems as a programming model:

- A simple programming language with an abstract syntax normal form corresponding to multiple assignment.
- Integrated specifications allowing verification and testing.
- A module system for reuse of components based on variable substitution.
- Additional module combinators corresponding to the logical connectives &, | and ⇒ (conjunction, disjunction and implication) with most distributivity properties intact.
- Rules for flattening module expressions.

Our semantics assumes an underlying programming language for code that computes variable valuations that satisfy individual constraints, but is completely independent of the choice of built-in types and primitive operations of that language.

The paper introduces quite a few definitions. On the one hand they define the semantics of constraint systems, which we apply in examples drawn from the domain of GUIs. On the other hand they build towards fitting constraint systems into the institution [13] notion. Institutions are a theoretical framework for understanding model-based logic, but they can also be used to explore metalevel design principles for programming languages, such as modular reuse mechanisms. Our semantic foundation is based on algebraic specification theory; see the book [14] for a comprehensive introduction. The specification language CASL [15], designed around the institution notion, and tool frameworks like Hets [16] show the flexibility that such a design offers. The history of algebraic specifications for computer science goes back to the 1970s, with the ADJ group [18,17] and what was to become the Larch group [19,20] as some influential examples. The two books [21,22] gave an important introduction to the field as of the late 1980s. The flexibility of algebraic specifications for reuse has been inspirational for our approach to MDCS semantics. Especially the role that institutions can play in reuse and modularisation has motivated our work.

The organisation of the paper is as follows. As motivation, Section 2 describes how constraint systems appear in GUIs. It also introduces a simple constraint system used in examples later in the paper. Section 3 defines the syntax and semantics of a multiple assignment programming language. Section 4 explores dataflow constraints and composition operators, the building blocks for multiway dataflow constraint systems. Section 5 relates the formalism to practical programming, applying the formalism to modelling constraint systems arising in GUIs. Section 6 gathers all the definitions and shows how dataflow constraints form an institution. Section 7 defines constraint systems with modular reuse. Section 8 summarises our findings.

2. Motivation: constraint systems in GUIs

We have found that complex user interface behaviours that are typically programmed in the "event handling logic", separately for each user interface, can be expressed as reusable algorithms when the state of a user interface is managed as a constraint system. Such behaviours include propagating values, enabling and disabling widgets, recording user interaction as scripts, visualising and controlling the dataflow, and undo [23,25,24].

As an example, consider the small GUI in Fig. 1. This GUI for resizing an image is a simplified version of dialogs found in many image manipulation applications. An image's initial width and height are determined at the launch of the dialog. The user can specify a new width and height relative to the initial values or directly as the number of pixels. Further, the user can request that the GUI preserves the initial aspect ratio.

The variables are dependent on each other: changing the value of one triggers changes in others so that the GUI returns to a consistent state. These dependencies and the set of consistent states are expressed by the constraint system in Fig. 2. The variables w_i, w_r, and w_a are bound, respectively, to the initial, relative, and absolute width fields and h_i, h_r, and h_a to the corresponding height fields. Fig. 2a shows the three constraints in the system: c_1 is for enforcing the relation $w_a = w_i * w_r$, c_2 the relation $h_a = h_i * h_r$, and c_3 the relation $w_r = h_r$. The "Keep aspect ratio" checkbox has no corresponding variable in the constraint system. Its value determines whether the c_3 constraint is active or not.

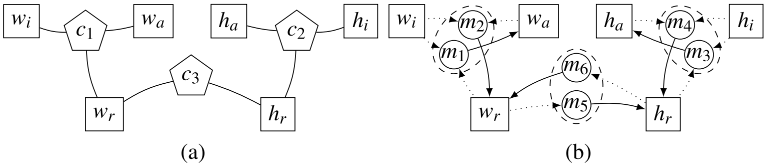


Fig. 2. The constraint system arising from the GUI of Fig. 1. Figure a depicts the relations (c_1, c_2, and c_3) amongst the variables in the system. Figure b shows the relations split into functional dependencies. A dashed ellipsis marks the set of methods that together implement a constraint's relation; for example, applying either of the methods m_3 or m_4 satisfies c_2.

Fig. 2b shows the three constraints' decomposed into constraint satisfaction methods; executing a method satisfies the relation of its constraint. Solving a *multiway dataflow constraint system* boils down to selecting one method from each constraint such that they can be executed in an order that does not invalidate already enforced constraints, and then executing the selected methods in such an order.

3. Syntax and semantics of a multiple assignment language

Here we define a programming model for multiway dataflow constraint systems as an *abstract programming language*, i.e., with abstract syntax and semantics relative to a base API. This follows the ideology suggested by the ADJ group [18] of not looking at the concrete syntax but focusing on the initial algebra of the syntax. In our case we can use the multiple assignment as the normal form for the syntax. The language builds on a base API (interface) that abstracts all primitive types and functions of the language, includes all standard library types and functions, and encompasses any domain specific extension needed by a user. In this way we achieve several exposition simplifications, yet a precise definition of the programming language.

- We avoid a lengthy exposition of grammar rules for the concrete syntax.
- We avoid deciding what types and operations are part of the language or library.
- We do not have to describe how to declare and implement new types and functions at the user level, since we can assume all such definitions are embodied in the base API.

Dealing with any of these issues becomes an orthogonal extension to the exposition of the abstract programming language.

As can be seen in the introductory example, constraints of a constraint system are connected by matching variable names. Thus, mechanisms for a module system based on variable matching are central to our presentation. These pieces are connected in Section 6.

In this section we first define standard notions of interfaces (signatures), expressions, and predicate expressions. Then we define a small programming language built on multiple assignment statements. This programming language is given a simple set theoretic semantics for an arbitrary model of the base interface.

In this exposition we need to reconcile established terminology and notation of algebraic specifications, programming languages, and constraint systems. We decided to use a terminology close to programming languages. Thus we use interface I where algebraic specifications use signature Σ, we separate expressions E_I (which algebraic specifications call terms) and statements S_I since both of these are terms in language theory, call an individual statement a method $m \in S_V$ (which also rings nicely with multiple assignment statement) to be compatible with constraint system terminology and avoid confusion with substitutions s, and we replace the term algebra A with the term model μ to avoid confusion with assignment sets $A_{\mu,V}$. We also use the unfamiliar adjective *staid* to introduce specific requirements on interfaces and models. The staid properties serve two purposes: to allow a uniform use of reasoning tools for expressions and predicates for dataflow constraint systems, and to provide an equivalence relation on methods (statements) which have multiple assignments as representatives—another benefit of the initial algebra approach to syntax.

Syntactic substitutions define mappings on variables, expressions and statements. We explore substitution as a means of adapting variable names for modular reuse, and as a means of formalising both multiple assignment and statement composition. In Section 6 we use substitutions to build a module system based on variable alignment in the institution setting.

3.1. Interfaces and expressions

Type correct expressions are constructed from typed function declarations and a set of typed variables, where the types and functions are declared by an interface.

Definition 3.1. An *interface* I declares a set of *types* $\mathrm{Typ}(I)$ and a set of *functions* $\mathrm{Fun}(I)$. A function $f \in \mathrm{Fun}(I)$ has an argument list $\mathsf{arg}(f) = (t_1, \dots, t_k) \in \mathrm{Typ}(I)^*$, $k \geq 0$, and a result type $\mathsf{res}(f) = t \in \mathrm{Typ}(I)$.

The notation X^* denotes all sequences of elements from the set X. As a shorthand, a function declaration can be written $f : t_1, \ldots, t_k \to t$.

Definition 3.2. A *staid interface* I is an interface with a type Predicate $\in \mathrm{Typ}(I)$ and functions

$$\begin{array}{rll}
\_==\_ : t, t \to & \text{Predicate} & \text{for all } t \in \mathrm{Typ}(I) \text{ (equality)},\\
(\_?\_:\_) : \text{Predicate}, t, t \to & t & \text{for all } t \in \mathrm{Typ}(I) \text{ (choice)},\\
\text{TRUE}, \text{FALSE} :\to & \text{Predicate} & \text{(truth values)},\\
!\_ : \text{Predicate} \to & \text{Predicate} & \text{(negation), and}\\
\_\&\_, \_|\_, \_=>\_ : \text{Predicate}, \text{Predicate} \to & \text{Predicate} & \text{(conjunction, disjunction, implication).}
\end{array}$$

A function with the result type Predicate is called a *predicate*.

Definition 3.3. A collection of *variables* V for an interface I declares a set of variable names $\mathrm{Nam}(V)$ and a function $\mathsf{typ}_V : \mathrm{Nam}(V) \to \mathrm{Typ}(I)$.

As a shorthand, we write $v \in V$ rather than $v \in \mathrm{Nam}(V)$, and we drop the subscript V on typ_V when it is unambiguous. The same goes for other set operations on variables, assuming that the typing functions are compatible. For instance, the subset relation $X \subseteq V$ holds when $\mathrm{Nam}(X) \subseteq \mathrm{Nam}(V)$ and $\mathsf{typ}_X(x) = \mathsf{typ}_V(x)$ for every $x \in \mathrm{Nam}(X)$. Let $\emptyset$ denote the empty collection of variables, i.e., $\mathrm{Nam}(\emptyset) = \{\}$.

Definition 3.4. The *expressions* of type $t \in \mathrm{Typ}(I)$ on an interface I with variables V is a set $E_{I,V,t}$ freely generated by

- for $v \in \mathrm{Nam}(V)$ with $\mathsf{typ}(v) = t$, then $v \in E_{I,V,t}$, and
- for $f : t_1, \ldots, t_k \to t \in \mathrm{Fun}(I)$, $e_1 \in E_{I,V,t_1}, \ldots, e_k \in E_{I,V,t_k}$, then $f(e_1, \ldots, e_k) \in E_{I,V,t}$.

In expressions we use the function name as a shorthand for the function declaration, tacitly assuming that all ambiguities are handled as needed. Let the set of all type correct expressions be $E_{I,V} = \cup_{t \in \mathrm{Typ}(I)} E_{I,V,t}$. The expressions $E_{I,\emptyset}$ are called *variable-free* or *ground expressions*. For an expression $e \in E_{I,V}$, we let $\mathsf{var}(e) \subseteq V$ be the set of variables that appear in e, and we extend the typing function from variables to $\mathsf{typ} : E_{I,V} \to \mathrm{Typ}(I)$ by $\mathsf{typ}(e) = t$ for $e \in E_{I,V,t}$.

Aligning methods on global variables is an important aspect of multiway dataflow constraint systems. We therefore introduce substitution as a mechanism to change variables in expressions.

Definition 3.5. Let I be an interface and X and Y be variables for I.

- A *substitution (on variables)* $s : X \to E_{I,Y}$ is a function from $\mathrm{Nam}(X)$ to $E_{I,Y}$ such that $\mathsf{typ}(x) = \mathsf{typ}(s(x))$ for all $x \in \mathrm{Nam}(X)$.
- If $s(x) \in Y$ for all $x \in X' \subseteq X$ then s is a *renaming limited to* X'. It is simply a *renaming* when it is a renaming for all of X.

There is no requirement that X and Y are disjoint. We do not impose any injectivity or surjectivity requirements on a renaming.

Given a set of n distinct variables $X = \{x_1, \ldots, x_n\}$, we can define a substitution $s : X \to E_{I,Y}$ by the list $[x_1 \mapsto e_1, \ldots, x_n \mapsto e_n]$, for $e_1, \ldots, e_n \in E_{I,Y}$, meaning $s(x_1) = e_1, \ \ldots, \ s(x_n) = e_n$.

A substitution on variables $s : X \to E_{I,Y}$ can be extended to a substitution on expressions, i.e., to a function $\overline{s} : E_{I,X} \to E_{I,Y}$ defined by

$$\overline{s}(e) = \begin{cases} s(e) & \text{for } e \in X, \\ f(\overline{s}(e_1), \ldots, \overline{s}(e_k)) & \text{for } e = f(e_1, \ldots, e_k). \end{cases} \tag{1}$$

Note that variable substitution aligns with expression substitution for variables, $s(x) = \overline{s}(x)$ for all $x \in X$. We therefore normally overload the notation using s for both.

Definition 3.6. Let X, Y and Z be variables for an interface I.

The composition of a substitution $r : X \to E_{I,Y}$ and $s : Y \to E_{I,Z}$ is a substitution $(s \circ r) : X \to E_{I,Z}$ defined by

$$(s \circ r)(x) = \overline{s}(r(x)). \tag{2}$$

Substitution composition is associative since function composition is associative. Define $\mathsf{id}_V : V \to E_{I,V}$ by $\mathsf{id}_V(v) = v$ for all variables $v \in V$, then id_V is neutral w.r.t. composition. That is, for $s : X \to E_{I,Y}$ then $\mathsf{id}_Y \circ s = s$ and $s \circ \mathsf{id}_X = s$. The composition of two renamings is a renaming, the composition of injective renamings is injective, the composition of surjective renamings is surjective, and the composition of bijective renamings is bijective.

Fact 3.7. Let V be variables for an interface I. Then the substitutions $s : V \to E_{I,V}$ form a monoid with substitution composition $\circ$ as the binary operation and the identity substituion $\mathsf{id}_V : V \to E_{I,V}$ as neutral element.

Predicate expressions should follow normal semantic conventions.

Definition 3.8. Let I be a staid interface with variables $x, y, z, x_1, \ldots, x'_1, \ldots \in V$, the types of which are clear from the context.

In a *staid model* for I the following properties hold

$$\begin{aligned}
\text{TRUE} &\neq \text{FALSE}, & (3)\\
(x == x) &= \text{TRUE}, & (4)\\
(x == y) = \text{TRUE} &\Leftrightarrow (y == x) = \text{TRUE}, & (5)\\
(x == y) = \text{FALSE} &\Leftrightarrow (y == x) = \text{FALSE}, & (6)\\
\big(x == y \mathbin{\&} y == z) = \text{TRUE} &\Rightarrow (x == z\big) = \text{TRUE}, & (7)\\
\big(x_1 == x'_1 \mathbin{\&} \ldots \mathbin{\&} x_k == x'_k\big) = \text{TRUE} &\Rightarrow \big(f(x_1, \ldots, x_k) == f(x'_1, \ldots, x'_k)\big) = \text{TRUE} & \\
&\text{for all } f : t_1, \ldots, t_n \to t \in \text{Fun}(I), & (8)\\
\text{TRUE} \mathbin{?} x : y &= x, & (9)\\
\text{FALSE} \mathbin{?} x : y &= y, & (10)\\
\text{TRUE} \mathbin{\&} \text{TRUE} &= \text{TRUE}, & (11)\\
\text{TRUE} \mathbin{\&} \text{FALSE} &= \text{FALSE}, & (12)\\
\text{FALSE} \mathbin{\&} \text{TRUE} &= \text{FALSE}, & (13)\\
\text{FALSE} \mathbin{\&} \text{FALSE} &= \text{FALSE}, & (14)\\
\text{TRUE} \mid \text{TRUE} &= \text{TRUE}, & (15)\\
\text{TRUE} \mid \text{FALSE} &= \text{TRUE}, & (16)\\
\text{FALSE} \mid \text{TRUE} &= \text{TRUE}, & (17)\\
\text{FALSE} \mid \text{FALSE} &= \text{FALSE}, & (18)\\
\text{TRUE} => \text{TRUE} &= \text{TRUE}, & (19)\\
\text{TRUE} => \text{FALSE} &= \text{FALSE}, & (20)\\
\text{FALSE} => \text{TRUE} &= \text{TRUE}, & (21)\\
\text{FALSE} => \text{FALSE} &= \text{TRUE}, & (22)\\
!\,\text{TRUE} &= \text{FALSE}, & (23)\\
!\,\text{FALSE} &= \text{TRUE}. & (24)
\end{aligned}$$

The first line states that the two constants TRUE, FALSE are different, the next 4 lines establish that $==$ is an equivalence relation (in the model), while Equation (8) adds the congruence property. The next two lines define the effect of the choice function. The rest are truth tables for the boolean connectives. From this we can, for a two-valued logic, derive the associative and commutative properties for $\&$ and $|$, their distributivity, $(p => q) = (!\,p \mid q)$, involutive properties for $!$, de Morgan's laws, $(p \mathbin{?} e_1 : e_2) = (!\,p \mathbin{?} e_2 : e_1)$, etc. From Equations (4)-(8) it follows that $==$ is a congruence for I. We also have that, interpreting $e_1 == c_2$ as a shorthand for $(e_1 == c_2) = \text{TRUE}$,

- when $e_1 == e_2$ then $s(e_1) == s(e_2)$ for expressions $e_1, e_2 \in E_{I,X}$ and substitution $s : X \to E_{I,Y}$, and
- when $e_1 == e'_1,\ \ldots, e_n == e'_n$, for n a natural number, then $[x_1 \mapsto e_1,\ \ldots, x_n \mapsto e_n](e) == [x_1 \mapsto e'_1,\ \ldots, x_n \mapsto e'_n](e)$ for expressions $e_1, e'_1, \ldots, e_n, e'_n \in E_{I,Y}$ and $e \in E_{I,X}$ and distinct variables $x_1, \ldots, x_n \in X$.

This approach is very similar to the approach taken in Larch and is supported by the Larch prover [19,20]. Note that $=$, $\Rightarrow$, $\Leftrightarrow$ are at the proof system level, thus putting restrictions on staid models. The "proof table" setup gives the normal semantics of predicates when generated in TRUE, FALSE, while admitting multivalued logics without any limiting constraints.

3.2. Multiple assignment statements

The basic building block for algorithms in our abstract programming language are sequences of multiple assignments and choice statements. We term such sequences *methods*, and treat them as our algorithmic units.

Definition 3.9. Let I be a staid interface and V variables for I.

The *statements* (or the *methods*) $S_{I,V}$ for interface I with variables V are freely generated by

- (multiple assignment) $x_1, \dots, x_n := e_1, \dots, e_n$ where $x_i \in V$, $e_i \in E_{I,V}$, all x_i are distinct, and $\text{Typ}(x_i) = \text{Typ}(e_i)$ for all i where $0 < i \leq n$,
- (skip) **skip**,
- (sequence) $m_1; m_2$ where $m_1, m_2 \in S_{I,V}$ are statements, and
- (choice) $\textbf{if}\, p\, \textbf{then}\, m_1\, \textbf{else}\, m_2\, \textbf{end}$ where $p \in E_{I,V,\text{Predicate}}$ and $m_1, m_2 \in S_{I,V}$ are statements.

Note the similarity between multiple assignment statements and substitutions. We will use this to translate between the two notions.

Definition 3.10. Let $m = (x_1, \dots, x_n := e_1, \dots, e_n)$ be a multiple assignment statement and $s = [x_1 \mapsto e_1, \dots, x_n \mapsto e_n]$ be a substitution for a staid interface I with distinct variables $x_1, \dots, x_n \in V$ and expressions $e_1, \dots, e_n \in E_{I,V}$.

Define $m^{\Phi} = s$ (met-to-sub) which takes a multiple assignment statement to its corresponding substitution, and $s^{\ominus} = m$ (sub-to-met) which takes a substitution to its corresponding multiple assignment statement.

The two operations are inverses, i.e., $m = (m^{\Phi})^{\ominus}$ and $s = (s^{\ominus})^{\Phi}$.

We consider an assignment $v := v$ to be vacuous, and that we can add/remove vacuous assignments in any multiple assignment statement, i.e., $(x_1, \dots, x_n := e_1, \dots, e_n) = (x_1, \dots, x_n, v := e_1, \dots, e_n, v)$ for $v \notin \{x_1, \dots, x_n\}$. The following links vacuous assignments with modifications of substitutions.

Definition 3.11. Let I be an interface, V, X, Y be compatible variables for I and $s : X \to E_{I,Y}$ be a substitution.

- The *V-extended substitution* $\lceil s \rceil^V : (X \cup V) \to E_{I,Y \cup V}$ is the substitution
$$\lceil s \rceil^V(v) = \begin{cases} s(v) & v \in X \\ v & \text{otherwise.} \end{cases}$$
- Define $m^{\Phi V} = \lceil m^{\Phi} \rceil^V$ as a shorthand to extend the substitution corresponding to a multiple assignment m with extra variables.
- The *V-stripped substitution* $\lfloor s \rfloor_V : X' \to E_{I,Y'}$, where the variables $X' = X \cap V$ and $Y' = \cup_{x \in X'} \text{var}(s(x)) \subseteq Y$, is the substitution $\lfloor s \rfloor_V(x) = s(x)$ for all $x \in X'$.
- The *trimmed substitution* $\lfloor s \rfloor = \lfloor s \rfloor_{V'}$, where $V' = \{x \in X \mid s(x) \neq x\}$.
- Define $s^{\lfloor \ominus \rfloor} = \lfloor s^{\ominus} \rfloor$ as a shorthand for trimming the multiple assignment statement corresponding to a substitution s.

Fact 3.12. Trimming a substitution $s : X \to E_{I,Y}$ for variables $X, Y \subseteq V$ removes all identity mappings from the substitution. Thus $\lfloor \text{id}_V \rfloor$ is the empty substitution, $\lfloor s \rfloor = \lfloor \lfloor s \rfloor \rfloor$, and $\lfloor s \rfloor = \lfloor \lceil s \rceil^V \rfloor$. If $X \cap Y = \emptyset$ then $\lceil s \rceil^V \circ \lceil s \rceil^V = \lceil s \rceil^V$ and $\lfloor s \rfloor = s$.

We call $X = \text{out}(m)$ the outputs and $Y = \text{inp}(m)$ the inputs of $m \in S_{I,V}$ for $m^{\Phi} : X \to E_{I,Y}$ and variables $X, Y \subseteq V$. The variables common to X and Y, $\text{upd}(m) = \text{inp}(m) \cap \text{out}(m)$, are called the updates of m. The variables of m are $\text{var}(m) = \text{inp}(m) \cup \text{out}(m)$. The method is *minimal*, if its corresponding substitution is trimmed. We implicitly refer to the minimal method when talking about its inputs and outputs.

In accordance with common semantic conventions for statements, we want the following properties for methods.

Definition 3.13. Let I be an interface, $x_1, \dots, x'_1, \dots \in V$ be variables for I, $e_1, \dots, e'_1, \dots \in E_{I,V}$ be expressions, and $p \in E_{I,V,\text{Predicate}}$ be a predicate expression.

In a *staid statement semantics* the following properties hold:

- Any permutation of assignments $x_1, \dots, x_n := e_1, \dots, e_n$ has the same semantics.
- Adding or removing a vacuous assignment from a multiple assignment statement has the same semantics as the original multiple assignment.

- **skip** has the same semantics as the vacuous assignment $\epsilon := \epsilon$, where ϵ is the empty list of variables.
- **if** p **then** $x_1, \ldots, x_n := e_1, \ldots, e_n$ **else** $x_1, \ldots, x_n := e'_1, \ldots, e'_n$ **end** has the same semantics as $x_1, \ldots, x_n := p\,?\,e_1 : e'_1, \ldots, p\,?\,e_n : e'_n$.
- Let

$$m = \big(x_1, \ldots, x_n := e_1, \ldots, e_n\big),$$
$$m' = \big(x'_1, \ldots, x'_n := e'_1, \ldots, e'_{n'}\big)$$

be multiple assignment statements. Then $m; m'$ has the same semantics as

$$x'_1, \ldots, x'_{n'}, \ldots, x_j, \ldots := m^{\Phi V}(e'_1), \ldots, m^{\Phi V}(e'_{n'}), \ldots, m^{\Phi V}(x_j), \ldots$$

where $x_j \in \{x_1, \ldots, x_n\} \setminus \{x'_1, \ldots, x'_{n'}\}$.

Note that a multiple assignment statement for variables $X \subseteq V$ can be expanded to a multiple assignment statement on all variables of V by adding vacuous assignments. This simplifies reasoning on composition of multiple assignment statements.

The staid semantics properties define a quotient on method syntax, with multiple assignment statements as (non-canonical) representatives for each equivalence class.

Proposition 3.14. *In staid statement semantics any statement is equivalent to a single multiple assignment statement.*

Proof. We take the claim as an induction hypothesis. By staid semantics, the **skip** statement is a multiple assignment. By the hypothesis, the statements in each branch of a choice statement can be coalesced into a single multiple assignment. By staid semantics, a choice statement with a multiple assignment in each branch can be replaced by a single multiple assignment statement. By staid semantics, a sequence of assignments can be replaced by a single assignment statement. □

For two multiple assignment statements m and m' and the multiple assignment statement m'' corresponding to their composition, we get that $\mathsf{out}(m'') = \mathsf{out}(m) \cup \mathsf{out}(m')$ and $\mathsf{var}(m'') = \mathsf{var}(m) \cup \mathsf{var}(m')$. In these and similar situations we often write $\mathsf{out}(m; m')$, $\mathsf{var}(m; m')$ etc. using $m; m'$ as a shorthand for introducing the multiple assignment statement m''.

Proposition 3.15 *(Contravariant composition rules for methods/substitution). Let m and m′ be two multiple assignment statements, then in staid statement semantics* $(m; m')^{\Phi V} = m^{\Phi V} \circ m'^{\Phi V}$.

Proof. Let the substitution $s = m^{\Phi V}$ and $s' = [\ldots, x'_j \mapsto e'_j, \ldots] = m'^{\Phi V}$. The composition $m; m'$ corresponds to the substitution $[\ldots, x'_j \mapsto s(e'_j), \ldots]$, which is the same as $[\ldots, x'_j \mapsto s(s'(x'_j)), \ldots] = s \circ s'$. □

Proposition 3.16. *For staid statement semantics the statement composition is associative with* **skip** *as the neutral statement.*

Proof. The composition of three multiple assignment statements m, m', m'' can be studied as the composition of the corresponding substitutions, $\big((m; m'); m''\big)^{\Phi V} = (m^{\Phi V} \circ m'^{\Phi V}) \circ m''^{\Phi V} = m^{\Phi V} \circ (m'^{\Phi V} \circ m''^{\Phi V}) = \big(m; (m'; m'')\big)^{\Phi V}$. □

This gives us a monoid structure on statements with statement composition as the binary operation. We see that extracting variables $\mathsf{out}, \mathsf{var} : S_{I,V} \to V$ becomes a monoid homomorphism from sequence of multiple assignment statements to sets of variables (with $\cup$ as the binary operation).

With the initial algebra approach to syntax and staid semantics, we can now use a single multiple assignment statement as the *semantic normal form* for any sequence of statements. We will typically choose either a trimmed assignment method $(m^{\Phi})^{\lfloor\ominus\rfloor}$ or an assignment method extended to all the variables $(m^{\Phi V})^{\ominus}$, though other representatives can be used when needed.

We have omitted loops from our language since method execution will be controlled by an outside planner or iterator. If iteration is needed within a method, we assume the library provides the needed facilities.

We can further extend substitution to act on multiple assignment statements, i.e., arbitrary methods presented in their normal form.

Definition 3.17. Let V, W be compatible variables for an interface I, $m \in S_{I,V}$ be a multiple assignment statement $m = (x_1, \ldots, x_n := e_1, \ldots, e_n)$ for $x_1, \ldots, x_n \in V$ and $e_1, \ldots, e_n \in E_{I,V}$, and the substitution $s : V \to E_{I,W}$ be an injective renaming for $\mathsf{out}(m) = \{x_1, \ldots, x_n\} \subseteq V$.

The *application* of s to m yields a multiple assignment statement $s(m) \in S_{I,W}$ defined as $s(x_1), \ldots, s(x_n) := s(e_1), \ldots, s(e_n)$.

Here the left hand side variables (from V) have to be replaced by variables (from W) by the substitution. The right hand side expressions are changed according to the normal application of a substitution on expressions.

We can extend substitution to work on statement sequences by applying it individually to each statement. Such an application is **not** interchangeable with staid statement semantics normalisation.

Example 3.18. Refer to Fig. 2, where we let $m_1 = (w_a := \lfloor w_i w_r \rfloor)$ and $m_4 = (h_r := h_a/h_i)$, thus the statement sequence $m_4; m_1$ has normal form $m'' = (h_r, w_a := h_a/h_i, \lfloor w_i w_r \rfloor)$. Define the substitution r_r such that $r_r(h_r) = r_r(w_r) = r$ and the identity everywhere else, which fixes the aspect ratio r to be the same for both width and height. Now $r_r(m_4); r_r(m_1) = (r := h_a/h_i); (w_a := \lfloor w_i r \rfloor)$ with normal form $(r, w_a := h_a/h_i, \lfloor w_i h_a/h_i \rfloor)$ which is very different from $r_r(m'') = (r, w_a := h_a/h_i, \lfloor w_i r \rfloor)$ or equivalently $(w_a, r := \lfloor w_i r \rfloor, h_a/h_i)$ since the order of the assignments is immaterial.

3.3. Semantics for expressions

Our semantic notion is that of heterogeneous or many-sorted algebras [14]. We interpret types as denoting sets and functions as denoting total set theoretic functions. This gives flexibility in choosing both the interface and a model for the interface, but such that the predicate type and a few predicate operations must have a staid interpretation. This flexibility allows us to work with any domain-specific API and associated semantics when solving practical problems. We just "plug in" the appropriate interface and algebra to become compatible with the selected problem domain. The syntactic and semantic framework has been defined with this in mind.

Definition 3.19. The *model* $\mu : I \to \mathbf{Set}$ of an interface I defines a set $\mu(t)$ for every $t \in \mathrm{Typ}(I)$ and a total function $\mu(f) : \mu(t_1) \times \cdots \times \mu(t_k) \to \mu(t)$ for every function $f : t_1, \ldots, t_k \to t \in \mathrm{Fun}(I)$.

Definition 3.20. A *standard model* μ for a staid interface I is a model for I where

$$\mu(\text{Predicate}) = \{\text{tt}, \text{ff}\}, \tag{25}$$

$$\mu(\text{TRUE}) = \text{tt}, \tag{26}$$

$$\mu(\text{FALSE}) = \text{ff}, \tag{27}$$

$$\mu(==)(x, y) = \begin{cases} \text{tt} & \text{when } x = y, \\ \text{ff} & \text{when } x \neq y, \end{cases} \tag{28}$$

$$\mu(\_?\_:\_)(p, y, z) = \begin{cases} y & \text{when } p = \text{tt}, \\ z & \text{when } p = \text{ff}, \end{cases} \tag{29}$$

$$\mu(\&)(p, q) = \begin{cases} \text{tt} & \text{when } p = q = \text{tt}, \\ \text{ff} & \text{when } p = \text{ff or } q = \text{ff}, \end{cases} \tag{30}$$

$$\mu(|)(p, q) = \begin{cases} \text{tt} & \text{when } p = \text{tt or } q = \text{tt}, \\ \text{ff} & \text{when } p = q = \text{ff}, \end{cases} \tag{31}$$

$$\mu(=>)(p, q) = \begin{cases} \text{tt} & \text{when } p = \text{ff or } q = \text{tt}, \\ \text{ff} & \text{when } p = \text{tt and } q = \text{ff}, \end{cases} \tag{32}$$

$$\mu(!)(p) = \begin{cases} \text{tt} & \text{when } p = \text{ff}, \\ \text{ff} & \text{when } p = \text{tt}, \end{cases} \tag{33}$$

and tt and ff are distinct values.

Any model isomorphic to the standard model above is also considered a standard model for a staid interface.

In algebraic semantics variables are placeholders for values. A variable can be allocated a value of the appropriate type.

Definition 3.21. An *allocation* $a : V \to \mu$ of values to variables V for an interface I with model μ defines a value $a(v) \in \mu(\mathrm{typ}(v))$ for all $v \in \mathrm{Nam}(V)$.

Define $A_{\mu,V} = \{a : V \to \mu\}$ to be the set of all allocations of values from μ to variables V.

Definition 3.22. Let μ be a model for the interface I and $a \in A_{\mu,V}$ be an allocation to variables V for I.

The evaluation $[\![\_]\!]_{\mu,a,t} : E_{I,V,t} \to \mu(t)$ of expressions of type $t \in \mathrm{Typ}(I)$ with allocation a is defined by

$$[\![e]\!]_{\mu,a,t} = \begin{cases} a(e) & \text{for } e \in V, \\ \mu(f)([\![e_1]\!]_{\mu,a,t_1}, \ldots, [\![e_k]\!]_{\mu,a,t_k}) & \text{for } e = f(e_1, \ldots, e_k) \text{ and } f : t_1, \ldots, t_k \to t. \end{cases} \tag{34}$$

We often omit the type information t from $\llbracket e \rrbracket_{\mu,a,t}$, writing just $\llbracket e \rrbracket_{\mu,a}$, since we always have that $t = \mathrm{typ}(e)$. Variable-free expressions $E_{I,\emptyset}$ represent all values of μ that we can denote. A model μ may contain undenotable values. For instance, relatively few of the mathematical real numbers are denotable.

Proposition 3.23. *For a staid interface the standard model is staid.*

Proof. Follows easily from the definition of staid expressions and the semantic properties above. □

We can now study semantics for substitutions.

Definition 3.24. Let $\mu : I \to \mathbf{Set}$ be a model and X, Y be variables for an interface I.

The *semantics of a substitution* $s : X \to E_{I,Y}$ is a contravariant mapping of allocations $\llbracket s \rrbracket_{\mu,\_} : A_{\mu,Y} \to A_{\mu,X}$ defined by $\big(\llbracket s \rrbracket_{\mu,\_}(b)\big)(x) = \llbracket s(x) \rrbracket_{\mu,b}$ for $b \in A_{\mu,Y}$ and $x \in X$.

Proposition 3.25. *Let I be an interface, X and Y be variables for I, $\mu : I \to$ **Set** be a model for I, $s : X \to E_{I,Y}$ be a substitution, and $b : Y \to \mu$ be an allocation.*

For any expression $e \in E_{I,X}$ we have that $\llbracket s(e) \rrbracket_{\mu,b} = \llbracket e \rrbracket_{\mu,\llbracket s \rrbracket_{\mu,b}}$.

Proof. Here $\llbracket s \rrbracket_{\mu,b} \in A_{\mu,X}$ is an allocation that gives all variables in e a value. The value for variable x is $\llbracket s \rrbracket_{\mu,b}(x) = \llbracket s(x) \rrbracket_{\mu,b}$, where $s(x)$ is the subexpression replacing x in e. Hence, for all variables $x \in X$, evaluating the subexpression $s(x)$ inside $s(e)$ for allocation b yields the same value as using $\llbracket s \rrbracket_{\mu,b}(x)$ directly in e. □

Fact 3.26. For substitutions $s_1 : X \to E_{I,Y}$ and $s_2 : Y \to E_{I,Z}$ the composition semantics is contravariant $\llbracket s_2 \circ s_1 \rrbracket_{\mu,\_} = \llbracket s_1 \rrbracket_{\mu,\_} \circ \llbracket s_2 \rrbracket_{\mu,\_}$.

This is evident since $\llbracket s_2(s_1(x)) \rrbracket_{\mu,c} = \llbracket s_1(x) \rrbracket_{\mu,\llbracket s_2 \rrbracket_{\mu,c}} = \llbracket x \rrbracket_{\mu,\llbracket s_1 \rrbracket_{\mu,\llbracket s_2 \rrbracket_{\mu,c}}} = \llbracket x \rrbracket_{\mu,\_}\big(\llbracket s_1 \rrbracket_{\mu,\llbracket s_2 \rrbracket_{\mu,c}}\big) = \llbracket x \rrbracket_{\mu,\_}\big(\llbracket s_1 \rrbracket_{\mu,\_}(\llbracket s_2 \rrbracket_{\mu,\_}(c))\big)$ $= \llbracket x \rrbracket_{\mu,\_}\big((\llbracket s_1 \rrbracket_{\mu,\_} \circ \llbracket s_2 \rrbracket_{\mu,\_})(c)\big) = \big(\llbracket s_1 \rrbracket_{\mu,\_} \circ \llbracket s_2 \rrbracket_{\mu,\_}(c)\big)(x)$ for any allocation $c \in A_{\mu,Z}$ and variable $x \in X$.

3.4. Semantics for statements

The standard semantics for imperative languages uses a *global store* indexed by locations [26]. The variables of the program denote locations, and the content of the variable is stored at the location. This allows the modelling of pointers, of data structures requiring more than one location, such as C style arrays, and other effects of a language design that allows aliasing. Even though our notion of methods are imperative in the sense that we assign values to variables in the multiple assignment statements, we do not need a global store in our setting. The reason is that we can model the association of variables to values by allocations as defined above. Rather than using a global store, we define the semantics of methods as allocation transformers.

Definition 3.27. Let $\mu : I \to \mathbf{Set}$ be a standard model for the staid interface I with variables V.

The semantics of a statement $m \in S_{I,V}$ is a transformation of allocations $\llbracket m \rrbracket_{\mu,\_} : A_{\mu,V} \to A_{\mu,V}$ where for allocation $a \in A_{\mu,V}$ and each method (see Definition 3.9)

$$\llbracket x_1, \ldots, x_n := e_1, \ldots, e_n \rrbracket_{\mu,a}(v) = \begin{cases} \llbracket e_1 \rrbracket_{\mu,a} & \text{when } v = x_1, \\ \vdots & \\ \llbracket e_n \rrbracket_{\mu,a} & \text{when } v = x_n, \\ a(v) & \text{otherwise,} \end{cases} \tag{35}$$

$$\llbracket \mathbf{skip} \rrbracket_{\mu,a}(v) = a(v), \tag{36}$$

$$\llbracket m_1 ; m_2 \rrbracket_{\mu,a}(v) = \llbracket m_2 \rrbracket_{\mu,\llbracket m_1 \rrbracket_{\mu,a}}(v), \tag{37}$$

$$\llbracket \mathbf{if}\, p\, \mathbf{then}\, m_1\, \mathbf{else}\, m_2\, \mathbf{end} \rrbracket_{\mu,a}(v) = \begin{cases} \llbracket m_1 \rrbracket_{\mu,a}(v) & \text{when } \llbracket p \rrbracket_{\mu,a} = \mu(\text{TRUE}), \\ \llbracket m_2 \rrbracket_{\mu,a}(v) & \text{when } \llbracket p \rrbracket_{\mu,a} = \mu(\text{FALSE}). \end{cases} \tag{38}$$

Fact 3.28. The semantics of statement sequencing is the covariant composition of the component allocation transformations, $\llbracket m_1 ; m_2 \rrbracket_{\mu,\_} = \llbracket m_2 \rrbracket_{\mu,\_} \circ \llbracket m_1 \rrbracket_{\mu,\_}$.

Due to the semantics of substitutions being contravariant to the substitutions, and the substitutions being contravariant to the methods, we end up with the semantics of a method and its associated substitution to be the same.

Proposition 3.29. *Let $m \in S_{I,V}$ be a method and $s : X \to E_{I,Y}$ be a substitution for staid interface I with a standard semantics μ and variables $X, Y \subseteq V$. Then the following holds:*

$$\llbracket m \rrbracket_{\mu,\_} = \llbracket m^{\Phi V} \rrbracket_{\mu,\_} : A_{\mu,V} \to A_{\mu,V}$$
$$\llbracket \lceil s \rceil^{V} \rrbracket_{\mu,\_} = \llbracket s^{\ominus} \rrbracket_{\mu,\_} : A_{\mu,V} \to A_{\mu,V}$$

Proof. Consider any $v \in V$ and any $a \in A_{I,V}$. For the first claim we have that

$$\llbracket m \rrbracket_{\mu,a}(v) = \llbracket m^{\Phi V}(v) \rrbracket_{\mu,a} = \llbracket m^{\Phi V} \rrbracket_{\mu,a}(v).$$

Similarly for the second claim, we have that

$$\llbracket \lceil s \rceil^{V} \rrbracket_{\mu,a}(v) = \llbracket \lceil s \rceil^{V}(v) \rrbracket_{\mu,a} = \llbracket s^{\ominus} \rrbracket_{\mu,a}(v) \quad \square$$

Fact 3.30. The semantics of statement sequencing is contravariant to the semantics of the composition of the corresponding substitution, i.e., for a method sequence $m_1; m_2$ we get that $\llbracket m_1; m_2 \rrbracket_{\mu,\_} = \llbracket m_1^{\Phi V} \circ m_2^{\Phi V} \rrbracket_{\mu,\_} : A_{I,V} \to A_{I,V}$.

This follows since $\llbracket m_1; m_2 \rrbracket_{\mu,\_} = \llbracket m_2 \rrbracket_{\mu,\_} \circ \llbracket m_1 \rrbracket_{\mu,\_} = \llbracket m_2^{\Phi V} \rrbracket_{\mu,\_} \circ \llbracket m_1^{\Phi V} \rrbracket_{\mu,\_} = \llbracket m_1^{\Phi V} \circ m_2^{\Phi V} \rrbracket_{\mu,\_}$ by Fact 3.26.

Proposition 3.31. *Let $\mu : I \to$* **Set** *be a standard model for the staid interface I with variables V.*
Then the semantics for methods $m \in S_{I,V}$ is staid.

Proof. The proof is by cases from the definition of staid statement semantics 3.13.

- Since all variables are distinct, the defined multiple assignment of new values to variables is independent of ordering of assignments within the statement.
- A missing variable assignment or a vacuous assignment both represent the identity allocation mapping, $\llbracket \epsilon := \epsilon \rrbracket_{\mu,a} = a = \llbracket x_1, \ldots, x_n := x_1, \ldots, x_n \rrbracket_{\mu,a}$ for all $a \in A_{\mu,V}$.
- The **skip** statement and the vacuous assignment both represent the identity allocation mapping, $\llbracket \textbf{skip} \rrbracket_{\mu,a} = a = \llbracket \epsilon := \epsilon \rrbracket_{\mu,a}$ for all allocations $a \in A_{\mu,V}$.
- $\llbracket \textbf{if}\, p\, \textbf{then}\, x_1, \ldots, x_n := e_1, \ldots, e_n\, \textbf{else}\, x_1, \ldots, x_n := e'_1, \ldots, e'_n\, \textbf{end} \rrbracket_{\mu,a}(v)$ is equal to $\llbracket x_1, \ldots, x_n := p\,?\,e_1 : e'_1, \ldots, p\,?\,e_n : e'_n \rrbracket_{\mu,a}(v)$ for all variables $v \in V$ and for all allocations $a \in A_{\mu,V}$.
- The semantics of sequences $m; m'$ of multiple assignments $m, m' \in S_{I,V}$, where $m' = (\ldots, x'_i, \ldots := \ldots, e'_i, \ldots)$, is

$$\begin{aligned}
\llbracket m; m' \rrbracket_{\mu,a}(v) &= \llbracket m' \rrbracket_{\mu,\llbracket m \rrbracket_{\mu,a}}(v) \\
&= \begin{cases} \ldots & \\ \llbracket e'_i \rrbracket_{\mu,\llbracket m \rrbracket_{\mu,a}} & \text{when } v = x'_i, \\ \ldots & \\ \llbracket m \rrbracket_{\mu,a}(v) & \text{otherwise} \end{cases} \\
&= \begin{cases} \ldots & \\ \llbracket e'_i \rrbracket_{\mu,(m^{\Phi V})_{\mu,a}} & \text{when } v = x'_i, \\ \ldots & \\ \llbracket m^{\Phi V} \rrbracket_{\mu,a}(v) & \text{otherwise} \end{cases} \\
&= \begin{cases} \ldots & \\ \llbracket m^{\Phi V}(e'_i) \rrbracket_{\mu,a} & \text{when } v = x'_i, \\ \ldots & \\ \llbracket m^{\Phi V}(v) \rrbracket_{\mu,a} & \text{otherwise, i.e., } v = x_j \notin \{\ldots, x'_i, \ldots\} \end{cases} \\
&= \llbracket \ldots, x'_i, \ldots, x_j, \ldots := \ldots, m^{\Phi V}(e'_i), \ldots, m^{\Phi V}(x_j) \ldots \rrbracket_{\mu,a},
\end{aligned}$$

  for all variables $v \in V$ and for all allocations $a \in A_{\mu,V}$. $\square$

3.5. Axioms and propositions for models

If a predicate expression is true for all allocations in a model, it is a "*property*" of the model. A property is an "*axiom*" if we think of it as a requirement on which models we will accept. It is a "*proposition*" of the model if it happens to hold in a chosen model. In principle we can prove the "propositions" from the "axioms" of a model. Technically "axioms" and "propositions" are the same, the only difference being the authors' perspective. We will use the term axiom except when we specifically want to emphasise that the property should be proved for a model. Axioms demand properties of models, without deciding exactly which model we are considering. The properties we defined for staid semantics of staid interfaces are axioms on the models.

Definition 3.32. Let I be a staid interface and V a collection of variables for I.

An *axiom* is a predicate expression $p \in E_{I,V,\text{Predicate}}$.

A model μ for I satisfies an axiom $p \in E_{I,V,\text{Predicate}}$, written $\mu \models_{I,V} p$, iff $\llbracket p \rrbracket_{\mu,a} = \mu(\text{TRUE})$ for all allocations $a \in A_{\mu,V}$.

For short we say that p holds for μ when meaning $\mu \models_{I,V} p$. We also write $q \models_{I,V,\mu} p$ for predicates p, q as a shorthand for whenever $\mu \models_{I,V} q$ then $\mu \models_{I,V} p$.

This definition gives us boolean expression logic as our specification formalism. Boolean expressions are part of our staid interface notation, and are common as expressions in programming languages. Such axioms can be tested by running them as normal code and providing test cases as input data for the predicate's variables (allocations in our exposition above) [27]. Algebraic specification theory identifies a sequence of increasingly more powerful logics leading up to predicate logic: equational logic, conditional equational logic, boolean expression (propositional) logic, first order predicate logic [14]. In this sequence, boolean expression logic is the most powerful logic that can easily be tested in a programming language setting.

Example 3.33. For a staid interface I with predicate $\leq : t, t \to$ Predicate, the following axioms on variables $\{x, y, z\}$ of type t define a total order on the type t: $x{\leq}x$ (reflexive), $x{\leq}y \ \&\ y{\leq}x => x == y$ (antisymmetric), $x{\leq}y \ \&\ y{\leq}z => x{\leq}z$ (transitive), and $x{\leq}y \ | \ y{\leq}x$ (connected).

Here we used the shorthand of providing a set of axioms $P \subseteq E_{I,V,\text{Predicate}}$ rather than explicitly writing one big axiom $\&_{p \in P}\, p$ (recall that in the standard model $\&$ is associative and commutative): $\big(\mu \models_{I,V} P\big) = \big(\forall (p \in P)(\mu \models_{I,V} p)\big) = \big(\mu \models_{I,V} (\&_{p \in P}\, p)\big)$.

Axioms can be used to verify properties of software if they convey enough information about the model. Axioms are resistant to variable substitution.

Proposition 3.34. *Let X, Y be variables for a staid interface I, $s : X \to E_{I,Y}$ a substitution, and μ a standard model for I. If an axiom $p \in E_{I,X,\text{Predicate}}$ holds in μ, then $s(p) \in E_{I,Y,\text{Predicate}}$ holds in μ.*

Proof. That $p \in E_{I,X,\text{Predicate}}$ holds in μ means that p holds for every allocation $a : X \to \mu$. Let $b : Y \to \mu$ be some allocation. Then, for all $e \in E_{I,Y}$ we have that $\llbracket s(e) \rrbracket_{\mu,b} = \llbracket e \rrbracket_{\mu,\llbracket s \rrbracket_{\mu,b}}$. Hence, $s(p)$ holds for b, since p holds for any allocation. This is the case for any $b : Y \to \mu$, thus $s(p)$ will hold in μ whenever p holds in μ. □

Thus whenever $\mu \models_{I,X} p$ holds then $\mu \models_{I,Y} s(p)$ holds.

Example 3.35. Here we show how to specify *semiring* (sometimes called *rig*) using the notions we have introduced for axioms. The interface can be written as

```
type T;
function mult ( a:T, b:T ) : T;
function one () : T;
function plus ( a:T, b:T) : T;
function zero () : T;
```

with axioms stating that:

- `mult` is a monoid with `one` as unit:
  $\texttt{mult}(\texttt{mult}(a, b), c) == \texttt{mult}(a, \texttt{mult}(b, c))$ and $\texttt{mult}(a, \texttt{one}()) == a$ and $\texttt{mult}(\texttt{one}(), a) == a$.
- `plus` is a commutative monoid with `zero` as unit:
  $\texttt{plus}(\texttt{plus}(a, b), c) == \texttt{plus}(a, \texttt{plus}(b, c))$ and $\texttt{plus}(a, \texttt{zero}()) == a$ and $\texttt{plus}(\texttt{zero}(), a) == a$ and $\texttt{plus}(a, b) == \texttt{plus}(b, a)$.
- `mult` distributes over `plus`:
  $\texttt{mult}(a, \texttt{plus}(b, c)) == \texttt{plus}(\texttt{mult}(a, b), \texttt{mult}(a, c))$ and
  $\texttt{mult}(\texttt{plus}(b, c), a) == \texttt{plus}(\texttt{mult}(b, a), \texttt{mult}(c, a))$.
- `mult` by `zero` annihilates:
  $\texttt{mult}(a, \texttt{zero}()) == \texttt{zero}()$ and $\texttt{mult}(\texttt{zero}(), a) == \texttt{zero}()$.

A semiring is commutative if `mult` is commutative: `mult` $(a,b) == $ `mult` (b,a). The standard example of a commutative semiring is the natural numbers: multiplication for `mult` with 1 for `one` and addition for `plus` with 0 for `zero`.

A semiring is idempotent if `plus` is idempotent: `plus` $(a,a) == a$. Examples of idempotent commutative semirings are the tropical semirings: addition for `mult` with 0 for `one` and minimum (or maximum) for `plus` with ∞ (respectively $-\infty$) for `zero`.

Idempotent commutative semirings also show up when we investigate dataflow constraint properties in Section 4.2.

4. Dataflow constraints

From now on we update our terminology to match the literature on multiway dataflow constraint systems. All interfaces will be staid with standard models (two valued logic), thus expression and statement semantics are staid.

The constraints $C_{I,V} = E_{I,V,\text{Predicate}}$ are the predicate expressions for interface I with variables V. A constraint $c \in C_{I,V}$ *holds* for an allocation $a \in A_{\mu,V}$, or a *satisfies* c, when $[\![c]\!]_{\mu,a} = \mu(\text{TRUE})$. We also write $A \models_{I,\mu,V} c$ when c holds for all $a \in A \subseteq A_{\mu,V}$

A method is a multiple assignment statement $m \in S_{I,V}$, or dually, a substitution $m^{\Phi V} : V \to E_{I,V}$ which is contravariant to the statement. The semantics of a method m is a covariant mapping of allocations, while the semantics of the corresponding substitution $m^{\Phi V}$ is a contravariant mapping of allocations, i.e., for a method sequence $m_1; m_2$ we have that $[\![m_1; m_2]\!]_{\mu,\_} = [\![m_1^{\Phi V} \circ m_2^{\Phi V}]\!]_{\mu,\_} : A_{\mu,V} \to A_{\mu,V}$. Note that in the following we use $m; m'$ consistently as a shorthand for the single multiple assignment statement corresponding to this composition in staid semantics (Proposition 3.14).

Definition 4.1. Let $c \in C_{I,V}$ be a constraint on variables V for the interface I, μ a model for I, and m a method with inputs X and outputs Y, for variables $X, Y \subseteq V$. The method m *satisfies* the constraint c in μ, written $m \models_{I,\mu,V} c$, iff $[\![c]\!]_{\mu,[\![m]\!]_{\mu,a}}$ holds for any allocation $a \in A_{\mu,V}$.

Note that satisfaction entails that m establishes c in one step. We also say that the constraint c *holds* for m in μ. We can stretch this to $M \models_{I,\mu,V} c$ for a set of methods M, meaning that c holds for every $m \in M$.

Fact 4.2. The satisfaction $m \models_{I,\mu,V} c$ is equivalent to $\mu \models_{I,V} m^{\Phi V}(c)$.

Both these definitions boil down to $[\![m^{\Phi V}(c)]\!]_{\mu,a}$ holding for all allocations $a \in A_{\mu,V}$. The fact basically elevates constraint satisfaction by a method to a "proposition" $m^{\Phi V}(c) == \text{TRUE}$ that should hold in the model. There is a rich literature and many tools, see for instance [20,16], for proving propositions from known properties (axioms) of the model.

Such a property can also be tested by choosing allocations (values) for the variables. Testing helps increase confidence in the correctness of the code m related to the constraint c. Randomly selecting allocations is often an efficient approach to thorough testing [28]. In some circumstances a careful selection of test cases amounts to proving a proposition [29].

Definition 4.3. Let μ be a model for an interface I. A *(multiway) dataflow constraint* is a tuple $d = \langle V, c, M\rangle$ where

- V are variables for I,
- $c \in C_{I,V}$ is a constraint, and
- $M \subseteq S_{I,V}$ is a set of methods $m \in M$ on V.

The *dataflow constraint is sound* for μ if $M \models_{I,\mu,V} c$.

Let var, con and met be the projections for dataflow constraints that return the variables V, the constraint c and the methods M, respectively. We denote by $D_{I,V}$ the set of all dataflow constraints with variables V for interface I. The sound dataflow constraints for a model μ for I are denoted by $D_{I,V,\mu} \subseteq D_{I,V}$.

Example 4.4. Assume the declarations and axioms for $\leq$ from Example 3.33. A dataflow constraint that maintains the relationship $x \leq y$ between two variables x and y is:

$$R = \langle \{x : t, y : t\}, x \leq y, \{x := y, y := x, \textbf{if}\, x{\leq}y\; \textbf{then skip else}\, x, y := y, x\, \textbf{end}\}\rangle.$$

The first two methods satisfy c because $x{\leq}x$ (reflexive), while the third (provided in a non-normalised form) satisfies c because $x{\leq}y \mid y{\leq}x$ (connected).

We mentioned above that the set of input and output variables of a method do not have to be mutually exclusive. The values that a method computes to its output variables can depend on the previous values of those variables. We do not assume that methods are idempotent. The following example demonstrates this.

Example 4.5. Assume a base API with integer operations and the integers as semantics. Define the dataflow constraint

$$\langle\{f_1 : \textbf{int}, f_2 : \textbf{int}\}, \text{TRUE}, \{f_1, f_2 := f_2, f_1 + f_2\}\rangle.$$

Executing the constraint's sole method repeatedly, from initial values $f_1 = 0, f_2 = 1$, leads to the variable f_1 to go through the Fibonacci numbers.

4.1. Dataflow constraint combinators

Dataflow constraint combinators allow us to combine dataflow constraints in the same way as we combine constraints: there are dataflow constraints for TRUE, FALSE, disjunction, sequencing, conjunction and implication. The combinators take into account combining sets of variables, combining the constraints and combining the methods of the involved dataflow constraints. The definitions below are not canonical, and other variations are possible. The combinators are defined to achieve many of the expected properties of logical combinators while being syntactically recognisable. For instance:

- When defining the disjunction of two dataflow constraints d_1, d_2, we take the disjunction of the two constraints $\mathsf{con}(d_1) \mid \mathsf{con}(d_2)$ and union of the two method sets $\mathsf{met}(d_1) \cup \mathsf{met}(d_2)$. We could have included also two-sided sequencing of the two method sets $\{(m_1; m_2), (m_2; m_1) \mid m_1 \in \mathsf{met}(d_1), m_2 \in \mathsf{met}(d_2)\}$, but then we would not achieve associativity of the disjunction combinator. If we decide to use one-sided sequencing $\{(m_1; m_2) \mid m_1 \in \mathsf{met}(d_1), m_2 \in \mathsf{met}(d_2)\}$ we loose commutativity.
- For sequencing of two dataflow constraints d_1, d_2 we take the conjunction of the two constraints $\mathsf{con}(d_1) \,\&\, \mathsf{con}(d_2)$ and the sequence of the two method sets $\mathsf{met}(d_1); \mathsf{met}(d_2) = \{(m_1; m_2) \mid m_1 \in \mathsf{met}(d_1), m_2 \in \mathsf{met}(d_2)\}$ when the output variable of m_2 does not modify $\mathsf{con}(d_1)$. We could have relaxed this to also allow $\big(m_1; \textbf{if}\, m_2^{\Phi}(\mathsf{con}(d_1))\, \textbf{then}\, m_2\, \textbf{else}\, \textbf{error}\, \textbf{end}\big)$. Here the lack of an alternative for the **else** part is written as **error**. Now at runtime the planner could backtrack when reaching an error, but then the runtime efficiency guarantees of planners will change. Or we could prove that the condition $m_2^{\Phi}(\mathsf{con}(d_1))$ will hold in the selected model, and include the unconditional $m_1; m_2$ in $\mathsf{met}(d_1; d_2)$, but this requires a powerful proof system as part of a software tool for the sequence combinator, giving unpredictable compile times for dataflow constraint combinators.
- We could introduce additional syntactic constraints on methods, e.g., that output and input variable sets always have to be disjoint. This will have the beneficial effect of making method sequencing idempotent $(m; m) = m$, but would disallow normalising sequences of methods to a single method. For instance, normalising the method sequence $x := y + 5; y := x + 4$ becomes impossible since $x, y := y + 5, y + 9$ would be illegal.

That said, studying planners and their syntactic requirements may allow us to come up with more specialised variants of the dataflow constraint combinators with interesting and useful properties.

The following two dataflow constraints play an important role in establishing useful algebraic properties of constraint system combinators.

Definition 4.6. The dataflow constraint $F = \langle\emptyset, \text{FALSE}, \emptyset\rangle$. The dataflow constraint $T = \langle\emptyset, \text{TRUE}, \{\epsilon := \epsilon\}\rangle$.

Definition 4.7. Let $d_1 = \langle V_1, c_1, M_1\rangle$ and $d_2 = \langle V_2, c_2, M_2\rangle$ be dataflow constraints for interface I with variables $V_1, V_2 \subseteq V$. We define the relation $d_1 \subseteq d_2$ on $D_{I,V}$ by $c_1 = c_2$ and $M_1 \subseteq M_2$.

Proposition 4.8. *Let V be variables for an interface I and $d_1, d_2 \in D_{I,V}$ be dataflow contraints such that $d_1 \subseteq d_2$.*
If d_2 is sound for a model μ for I, then d_1 will also be sound for μ.

Proof. It is clear that d_1 is a dataflow constraint with fewer methods than d_2, but each method in d_1 is also in d_2. Since every method in d_2 satisfies the soundness criterion, so will every method in d_1. □

In general, if we remove some methods from a sound dataflow constraint it will still be sound.

We will now define how to combine dataflow constraints and adapt them to different use contexts.

Definition 4.9. Let $d = \langle V, c, M\rangle \in D_{I,V}$ be a dataflow constraint for interface I with variables V and W, and $s : V \to E_{I,W}$ be a substitution.

Then a method $m \in M$ is *substitution safe* for s when (1) s is an injective renaming on $\mathsf{out}(m)$, i.e., $s(\mathsf{out}(m)) \subseteq W$ and for every $x, y \in V$ it is the case the $s(x) = s(y) \Rightarrow x = y$, and (2) s is interjection safe meaning the resulting method $s(m)$ does not unintentionally modify variables being used by $s(m)$ and $s(c)$, i.e., $\mathsf{var}\big(s((\mathsf{var}(m) \cup \mathsf{var}(c)) \setminus \mathsf{out}(m))\big) \cap s(\mathsf{out}(m)) = \emptyset$.

Define $s(d) = \langle W, s(c), M'\rangle$ where $M' = \{s(m) \mid m \in M$ and m is substitution safe for $s\}$.

It is obvious that $s(d) \in D_{I,W}$ is a dataflow constraint.

Theorem 4.10 *(Safe substitution preserves soundness). Let I be an interface, $d = \langle V, c, M\rangle$ a dataflow constraint, $s : V \to E_{I,W}$ a substitution, and V, W be variables for I.*

If d is sound for a model μ for I, then $s(d)$ is also a sound dataflow constraint for μ.

Proof. For a method $s(m) \in M'$ there are two cases to consider: (1) variables $v \in \mathsf{out}(m)$ that are being assigned to in m, $\dots, v, \dots := \dots, e, \dots$ and (2) variables $x \in (\mathsf{var}(m) \cup \mathsf{var}(c)) \setminus \mathsf{out}(m)$ that are not being assigned to. Both v and x are possibly appearing on the right hand side of m and possibly used in c. The substitution will map variables v to variables $w = s(v)$, and map x to expressions $s(x)$ and expressions e to expressions $s(e)$. So in $s(c)$ we will have w in place of v and $s(x)$ in place of x, and $s(m)$ will have the form $\dots, w, \dots := \dots, s(e), \dots$. We know that $m^{\Phi V}(c)$, where v has been replaced by e, holds in μ, and thus $s(m^{\Phi V}(c))$, where v has been replaced by $s(e)$ and x has been replaced by $s(x)$, holds in μ. In $s(m)^{\Phi V}(s(c))$, v has been replaced by w that is replaced by $s(e)$ and x has been replaced by $s(x)$. Since m is substitution safe for s, the substitution $s(m)^{\Phi V}$ does not capture any variable in $s(x)$, thus $s(m)^{\Phi V}(s(c)) = s(m^{\Phi V}(c))$. Hence $s(m)^{\Phi V}(s(c))$ holds in μ, as claimed. □

In other words, $\mathsf{met}(d) \models_{I,\mu,V} c$ implies $\mathsf{met}(s(d)) \models_{I,\mu,W} s(c)$. The substitution theorem allows us to reuse existing dataflow constraints in new contexts. Here we remove methods rather than place conditions on s. If s is very well behaved, i.e., it only renames the output variables of the methods in M, then none of the methods will be removed.

Proposition 4.11 *(Disjunction). Let μ be a model for an interface I with variables V, and $d_1 = \langle V_1, c_1, M_1\rangle$ and $d_2 = \langle V_2, c_2, M_2\rangle$ be dataflow constraints, such that $V_1, V_2 \subseteq V$. Define $d_1|d_2 = \langle V_1 \cup V_2, c_1|c_2, M_1 \cup M_2\rangle$.*

If both d_1 and d_2 are sound in μ, then $d_1|d_2$ is a sound dataflow constraint in μ.

Proof. The requirement for soundness is that any method establishes the constraint in one execution. This is clearly the case since the methods of d_1 establish c_1 and hence $c_1|c_2$, and similarly for the methods of d_2. □

Proposition 4.12 *(Sequencing). Let μ be a model for an interface I with variables V, and $d_1 = \langle V_1, c_1, M_1\rangle$ and $d_2 = \langle V_2, c_2, M_2\rangle$ be dataflow constraints, such that $V_1, V_2 \subseteq V$. Define $d_1; d_2 = \langle V_1 \cup V_2, c_1\&c_2, M'\rangle$ where M' is the set of methods $m_1; m_2$ such that $\mathsf{out}(m_2) \cap \mathsf{var}(c_1) = \emptyset$, for all $m_1 \in M_1$ and $m_2 \in M_2$.*

If both d_1 and d_2 are sound in μ, then $d_1; d_2$ is a sound dataflow constraint in μ.

Note that we only allow methods in M' where the second method does not modify the satisfaction of the first constraint.

Proof. The methods of $m_1; m_2 \in M'$ are such that m_1 first establishes c_1, then m_2 establishes c_2 without modifying any values that are involved in c_1. Thus $m_1; m_2$ establishes $c_1\&c_2$ as required. □

Proposition 4.13 *(Conjunction). Let μ be a model for an interface I with variables V, and $d_1 = \langle V_1, c_1, M_1\rangle$ and $d_2 = \langle V_2, c_2, M_2\rangle$ be dataflow constraints, such that $V_1, V_2 \subseteq V$. Define $d_1\&d_2 = (d_1; d_2)|(d_2; d_1)$.*

If both d_1 and d_2 are sound in μ, then $d_1\&d_2$ is a sound dataflow constraint in μ.

Proof. The methods from $d_1; d_2$ establish $c_1\&c_2$, and the methods from $d_2; d_1$ establish $c_2\&c_1$. The union of these methods thus establish the disjunction of the constraint, as required. □

Because $(c_1\&c_2)|(c_2\&c_1) = c_1\&c_2$, the conjunctive combinator establishes the same constraint as the sequence combinator. It just provides a larger set of methods, since it combines the methods of $m_1 \in M_1$ and $m_2 \in M_2$ in both sequences $m_1; m_2$ and $m_2; m_1$. The former sequence only includes method combinations where $\mathsf{out}(m_2) \cap \mathsf{var}(c_1) = \emptyset$ and the latter only combinations where $\mathsf{out}(m_1) \cap \mathsf{var}(c_2) = \emptyset$.

Proposition 4.14 *(Implication). Let μ be a model for an interface I with variables V, and $d_1 = \langle V_1, c_1, M_1\rangle$ and $d_2 = \langle V_2, c_2, M_2\rangle$ be dataflow constraints, such that $V_1, V_2 \subseteq V$. Define $(d_1 \Rightarrow d_2) = \langle V_1 \cup V_2, c_1 => c_2, M_2 \cup (M_1; M_2) \cup M'\rangle$ where $M_1; M_2 = \{m_1; m_2 \mid m_1 \in M_1, m_2 \in M_2\}$ and $M' = \{\textbf{if}\, c_1\, \textbf{then}\, m_2\, \textbf{else skip end} \mid m_2 \in M_2\}$.*

If both d_1 and d_2 are sound in μ, then $d_1 => d_2$ is a sound dataflow constraint in μ.

Proof. First note that if c_2 holds then $c_1 => c_2$ holds, irrespective of the value of c_1. Thus all methods of M_2 make the implication hold. Second note that all methods of M_1 makes c_1 hold, so they need to be followed by some method from M_2 to make c_2 and thus the implication hold. Thus all methods of $M_1; M_2$ make the implication hold. Third note that if c_1 holds a method from M_2 must be executed, while if c_1 does not hold, there is no need to update any variables. Thus all methods of M' make the implication hold. □

If we had a way of making $!\,c_1$ hold, say a dataflow constraint $d_3 = \langle V_1, !\,c_1, M'' \rangle$, then we could add M'' to the set of methods that makes $c_1 => c_2$ hold since they invalidate the premise c_1. We are unable in general to automatically take a method that establishes c and create a method that establishes $!\,c$, and thus will not provide a negation combinator for dataflow constraints.

We now have matching combinators for all the positive logical connectives.

Example 4.15. From the dataflow constraint R in Example 4.4 we can construct a dataflow constraint on three variables. Let $s = [x \mapsto y, y \mapsto z]$ be a substitution. Applying the substitution to R gives us the dataflow constraint $s(R) = \langle \{y : t, z : t\}, y \leq z, \{(y := z), (z := y), \big(y, z := y{\leq}z\,?\,y\,:\,z, y{\leq}z\,?\,z\,:\,y\}\big)\rangle$ after normalising the methods to the multiple assignment format.

$$
\begin{aligned}
R; s(R) = \langle &\{x : t, y : t, z : t\}, x \leq y \,\&\, y \leq z,\\
&\{(x, z := y, y), (y, z := x, x), (x, y, z := x{\leq}y\,?\,x\,:\,y, x{\leq}y\,?\,y\,:\,x, y)\}\rangle,\\
s(R); R = \langle &\{x : t, y : t, z : t\}, y \leq z \,\&\, x \leq y,\\
&\{(x, y := z, z), (x, z := y, y), (x, y, z := y, y{\leq}z\,?\,y\,:\,z, y{\leq}z\,?\,z\,:\,y)\}\rangle\\
s(R)\&R = \langle &\{x : t, y : t, z : t\}, y \leq z \,\&\, x \leq y,\\
&\{(x, y := z, z), (x, z := y, y), (y, z := x, x),\\
&\;(x, y, z := x{\leq}y\,?\,x\,:\,y, x{\leq}y\,?\,y\,:\,x, y), (x, y, z := y, y{\leq}z\,?\,y\,:\,z, y{\leq}z\,?\,z\,:\,y)\}\rangle.
\end{aligned}
$$

Example 4.16. Here we show an example with a conjunction between two constraints that are inconsistent. Assume we have a type t with arithmetic and comparison operations.

$$
\begin{aligned}
d_1 &= \langle \{x : t, y : t\}, x \leq y, \{(x := y), (y := x), (x := y - 1), (\mathbf{if}\, x > y\, \mathbf{then}\, x, y := y, x\, \mathbf{else\ skip\ end})\}\rangle,\\
d_2 &= \langle \{x : t, y : t\}, x > y, \{(y := x - 1), (\mathbf{if}\, x \leq y\, \mathbf{then}\, x, y := y, x - 1\, \mathbf{else\ skip\ end})\}\rangle
\end{aligned}
$$

Let the global system have the variables x, y. The conjunction of these constraints yields

$$
d_1 \& d_2 = \langle \{x : t, y : t\}, x{\leq}y \& x > y, \{\}\rangle.
$$

Given normal integer semantics for t there is obviously no way of satisfying the combined constraint. The set of methods also reduce to the empty set since all output variables of the methods are used in the constraint. Thus soundness has been preserved by the conjunction.

Example 4.17. This is a variation of the above example, but here we use a substitution to achieve a consistent constraint when combining the two dataflow constraints. Assume we have the same type t and the same two dataflow constraints as above. Let the global system have the variables $a, b : t$ and let $s = [x \mapsto a, y \mapsto b]$ and $s' = [x \mapsto b, y \mapsto a]$ be substitutions into the global variables $a, b : t$. The conjunction of the two constraints with their respective substitutions yields

$$
s(d_1) \& s'(d_2) = \langle \{a : t, b : t\}, a{\leq}b \& b > a, \{\}\rangle.
$$

Again we get an empty set of methods which ensures preservation of soundness. In this case, with standard integer semantics for t, we can actually satisfy the combined constraint, though our syntactic combinator rules cannot take this into account.

However, if we apply the implication combinator, we get a more interesting result.

$$
\begin{aligned}
&s(d_1) => s'(d_2)\\
&= \langle \{a : t, b : t\}, a{\leq}b => b > a, \{\\
&\quad (a := b - 1), (\mathbf{if}\, b \leq a\, \mathbf{then}\, b, a := a, b - 1\, \mathbf{else\ skip\ end}),\\
&\quad (a := b; a := b - 1),\\
&\quad (b := a; a := b - 1),\\
&\quad (a := b - 1; a := b - 1),\\
&\quad (\mathbf{if}\, a > b\, \mathbf{then}\, a, b := b, a\, \mathbf{else\ skip\ end}; a := b - 1),\\
&\quad (a := b; \mathbf{if}\, b \leq a\, \mathbf{then}\, b, a := a, b - 1\, \mathbf{else\ skip\ end}),\\
&\quad (b := a; \mathbf{if}\, b \leq a\, \mathbf{then}\, b, a := a, b - 1\, \mathbf{else\ skip\ end}),
\end{aligned}
$$

$$\begin{aligned}
&(a := b-1; \textbf{if}\, b \le a\, \textbf{then}\, b, a := a, b-1\, \textbf{else skip end}),\\
&(\textbf{if}\, a > b\, \textbf{then}\, a, b := b, a\, \textbf{else skip end}; \textbf{if}\, b \le a\, \textbf{then}\, b, a := a, b-1\, \textbf{else skip end}),\\
&(\textbf{if}\, a{\le}b\, \textbf{then}\, a := b-1\, \textbf{else skip end}),\\
&(\textbf{if}\, a{\le}b\, \textbf{then if}\, b \le a\, \textbf{then}\, b, a := a, b-1\, \textbf{else skip end else skip end}).\}\rangle.
\end{aligned}$$

Here we have three groups of methods: those from $s'(d_2)$, those from $s(d_1); s'(d_2)$, and finally the methods from $s'(d_2)$ checked by the premise $a{\le}b$ from $s(d_1)$. With axioms giving properties of the model and proof support tools, we might analyse what is going on in more detail and simplify the resulting dataflow constraint system.

For some specific constraints for an interface I and variables V, such as the constraint $c = (x_1 == e_1 \,\&\, \ldots \,\&\, x_n == e_n) \in C_{I,V}$, where $e_1, \ldots, e_n \in E_{I,Y}$ and $X, Y \subseteq V$ for $X = \{x_1, \ldots, x_n\}$, it is possible to generate a method $m = (x_1, \ldots, x_n := e_1, \ldots e_n)$. This is, however, a one-way dataflow method from variables Y to variables X, which does not give methods for dataflows into Y. Given specifications for I, e.g., that the expressions $e_1, \ldots, e_n$ are affine, it is possible to generate more methods. Exploring such possibilities builds on specifications with support tools. Though an interesting research direction, it is beyond the scope of this paper.

4.2. Dataflow constraint combinator properties

Here we look into properties of dataflow constraints, starting with how dataflow constraints interact with substitutions. Recall (Fact 3.7) that substitutions form a monoid with substitution composition $\circ$ as the binary associative operation and the identity substitution as the neutral element.

Proposition 4.18 *(Substitution is a monoid action on dataflow constraints). Let I be an interface, V variables for I, $s_1, s_2 : V \to E_{I,V}$ be substitutions and $d \in D_{I,V}$ a dataflow constraint. Then the following monoid action properties hold for substitutions applied to dataflow constraints.*

- *The identity substitution $\mathsf{id}_V : V \to V$ is neutral on dataflow constraints, i.e., $\mathsf{id}_V(d) = d$.*
- *Substitution composes on dataflow constraints, i.e., $s_2(s_1(d)) = (s_2 \circ s_1)(d)$.*

The monoid action preserves soundness for d.

Proof. Most of the needed properties follow from the definition of the identity substitution and of substitution composition. We need to show that the methods filtered by substitution safety in the second bullet point are $\mathsf{met}\big((s_2 \circ s_1)(d)\big) = \mathsf{met}\big(s_2(s_1(d))\big)$.

On both sides the injective renaming property gives the same filter, ensuring that the variables $\mathsf{out}(m)$ are renamed to distinct variables $\mathsf{out}(s_2(s_1(m)))$. The interjection safety criterion, for every $m \in \mathsf{met}(d)$, on the left of the equation above expands to $\mathsf{var}\big(s_1((\mathsf{var}(m) \cup \mathsf{var}(c)) \setminus \mathsf{out}(m))\big) \cap s_1\big(\mathsf{out}(m)\big) = \emptyset$ and $\mathsf{var}\big(s_2((\mathsf{var}(s_1(m)) \cup \mathsf{var}(s_1(c))) \setminus \mathsf{out}(s_1(m)))\big) \cap \big(s_2(\mathsf{out}(s_1(m)))\big) = \emptyset$, while on the right side of the equation expands to $\mathsf{var}\big((s_2 \circ s_1)((\mathsf{var}(m) \cup \mathsf{var}(c)) \setminus \mathsf{out}(m))\big) \cap \big((s_2 \circ s_1)(\mathsf{out}(m))\big) = \mathsf{var}\big(s_2(s_1((\mathsf{var}(m) \cup \mathsf{var}(c)) \setminus \mathsf{out}(m)))\big) \cap \big(s_2(s_1(\mathsf{out}(m)))\big) = \mathsf{var}\big(s_2((\mathsf{var}(s_1(m)) \cup \mathsf{var}(s_1(c))) \setminus \mathsf{out}(s_1(m)))\big) \cap \big(s_2(\mathsf{out}(s_1(m)))\big) = \emptyset$. This matches the second safety criterion from the left side. Note that if the first safety criterion on the left is violated, so will the second criterion: the injective renaming aspect of s_2 on $\mathsf{out}(s_1(m))$ will propagate the interjected variables from $\mathsf{var}\big(s_1((\mathsf{var}(m) \cup \mathsf{var}(c)) \setminus \mathsf{out}(m))\big)$, which will then not be disjoint from $\big(s_2(\mathsf{out}(s_1(m)))\big)$.

Since substitution preserves soundness, the monoid action also preserves soundness. □

Proposition 4.19 *(Substitution distributes over dataflow constraint combinators). Let V, W be variables for I, $s : V \to E_{I,W}$ be a substitution and $d_1, d_2 \in D_{I,V}$ be dataflow constraints. Then the following distribution properties of substitution over dataflow constraint combinators hold for any staid semantics.*

- *$s(d_1|d_2) = s(d_1)|s(d_2)$*
- *$s(d_1; d_2) \subseteq s(d_1); s(d_2)$*
- *$s(d_1\&d_2) \subseteq s(d_1)\&s(d_2)$*
- *$s(d_1 => d_2) \subseteq s(d_1) => s(d_2)$.*

The distribution properties also preserve soundness.

Proof. Let $V_1, V_2 \subseteq V$ be variables, $d_1 = \langle V_1, c_1, M_1\rangle$ and $d_2 = \langle V_2, c_2, M_2\rangle$ be dataflow constraints, and $W_1 = \mathsf{var}(s(V_1))$ and $W_2 = \mathsf{var}(s(V_2))$.

For all the combinators $\odot \in \{|, ;, \&, =>\}$, variables $\mathsf{var}(s(\mathit{var}(d_1 \odot d_2))) = \mathsf{var}(s(V_1 \cup V_2)) = W_1 \cup W_2 = \mathsf{var}(s(V_1)) \cup \mathsf{var}(s(V_2)) = \mathsf{var}(s(d_1)) \cup \mathsf{var}(s(d_2)) = \mathsf{var}(s(d_1) \odot s(d_2))$.

For all the combinators $\odot \in \{|, \&, \;=>\;\}$, constraints $\mathsf{con}(s(d_1 \odot d_2)) = s(\mathsf{con}(d_1 \odot d_2)) = s(\mathsf{con}(d_1) \odot \mathsf{con}(d_2)) = s(\mathsf{con}(d_1)) \odot s(\mathsf{con}(d_2))$.

For the disjunction combinator | the new set of methods are $M_1 \cup M_2$. The safety criterion filtering is thus the same for both $s(d_1|d_2)$ and $s(d_1)|s(d_2)$.

For the sequencing combinator; we need to consider two filtering mechanisms. In $s(m_1; m_2)$, for $m_1 \in M_1, m_2 \in M_2$, we know that all cases where m_2 modifies the constraint c_1 are filtered out, then the methods are normalised to multiple assignment and the substitution safety requirement is applied. In $s(m_1); s(m_2)$, for $m_1 \in M_1, m_2 \in M_2$, we know that first the substitution safety condition filters away methods from M_1 and M_2, before the sequencing constraint filters out those $s(m_2)$ that modify $s(c_1)$. Since s is an injective renaming on $\mathsf{out}(m_2)$ the latter filtering amounts to former sequence constraint filtering. The methods m_1 and m_2 can fail the substitution safety criterion for s for two reasons: (1) s is not an injective renaming on $\mathsf{out}(m_1) \cup \mathsf{out}(m_2)$ but then s will not be an injective renaming on the normalised method $m_1; m_2$, or (2) s interjects new output variables from $s(m_1)$ into its right hand side expressions or into c_1 or similarly for $s(m_2)$ and $s(c_2)$ but in either case applying s to the normalised method $m_1; m_2$ will violate the interjection criterion. Thus $s(d_1; d_2) \subseteq s(d_1); s(d_2)$.

The remaining combinators, $\odot \in \{\&, \;=>\;\}$ create the new set of methods from unions of sequences, which behaves similarly to the sequencing operator for the reated sets. In addition $=>$ adds a conditional method which adds $\mathsf{var}(c_1)$ to the inerjection criterion for $s(d_1 => d_2)$ but not for $s(d_1) => s(d_2)$. Thus $s(d_1 \odot d_2) \subseteq s(d_1) \odot s(d_2)$.

All the combinators and substitution preserve soundness individually, and hence in combination. □

From Example 3.18 we know there exists m_1 and m_2 which individually satisfy the substitution safety criterion, but where the normalised method $m_1; m_2$ does not. Hence we do not get an equality between $s(d_1; d_2)$ and $s(d_1); s(d_2)$.

Proposition 4.20 *(Disjunction combinator forms an idempotent commutative monoid). Let d_1, d_2 and d_3 be dataflow constraints in $D_{I,V}$. Then the following idempotent commutative monoid properties hold in staid semantics.*

- $F|d_1 = d_1$
- $d_1|F = d_1$
- $(d_1|d_2)|d_3 = d_1|(d_2|d_3)$
- $d_1|d_2 = d_2|d_1$
- $d_1|d_1 = d_1$

These properties also preserve soundness.

Proof. Seen by expanding the components of the combined dataflow constraints on both sides of the equality. Note that the individual operations preserve soundness on both sides of the equality. □

Proposition 4.21 *(Sequencing combinator forms a monoid with annihilation). Let d_1, d_2 and d_3 be dataflow constraints in $D_{I,V}$. Then the following monoid properties with annihilation hold in staid semantics.*

- $T; d_1 = d_1$
- $d_1; T = d_1$
- $(d_1; d_2); d_3 = d_1; (d_2; d_3)$
- $F; d_1 = F$
- $d_1; F = F$

These properties also preserve soundness.

Proof. Seen by expanding the components of the combined dataflow constraints on both sides of the equality. Note that the individual operations preserve soundness on both sides of the equality. □

Proposition 4.22 *(Conjunction combinator forms a commutative monoid with annihilation). Let d_1, d_2 and d_3 be dataflow constraints in $D_{I,V}$. Then the following commutative monoid with annihilation properties hold in staid semantics.*

- $T\&d_1 = d_1$
- $d_1\&T = d_1$
- $(d_1\&d_2)\&d_3 = d_1\&(d_2\&d_3)$
- $d_1\&d_2 = d_2\&d_1$
- $F\&d_1 = F$
- $d_1\&F = F$

These properties also preserve soundness.

Proof. Seen by expanding the components of the combined dataflow constraints on both sides of the equality. Note that the individual operations preserve soundness on both sides of the equality. □

Proposition 4.23 *(Distributivity of sequencing and conjunction combinators over disjunction combinator). Let d_1, d_2 and d_3 be dataflow constraints in $D_{I,V}$. Then the following distributivity properties hold in staid semantics.*

- $(d_1|d_2); d_3 = (d_1; d_3)|(d_2; d_3)$
- $d_1; (d_2|d_3) = (d_1; d_2)|(d_1; d_3)$
- $(d_1|d_2)\&d_3 = (d_1\&d_3)|(d_2\&d_3)$

The distributivity properties also preserve soundness.

Proof. Seen by expanding the components of the combined dataflow constraints on both sides of the equality. Note that the individual operations preserve soundness on both sides of the equality. □

This establishes similar combinator laws for dataflow constraints as we have for logical combinators. The major exception is the distributivity of conjunction over disjunction, $(d_1\&d_2)|d_3 = (d_1|d_3)\&(d_2|d_3)$, which does not hold.

The following two facts relate dataflow combinators to the semiring concept 3.35.

Fact 4.24 *(The sequencing combinator gives an idempotent semiring).* The dataflow combinators in staid semantics define an idempotent semiring by sequencing for `mult` with T for `one` and disjunction for `plus` with F for `zero`.

Fact 4.25 *(The conjunction combinator gives an idempotent commutative semiring).* The dataflow combinators in staid semantics define an idempotent commutative semiring by conjunction for `mult` with T for `one` and disjunction for `plus` with F for `zero`.

On a case by case basis we may come up with more combinators than those above. For instance, assume two dataflow constraints $d_1 = \langle V_1, c, P_1\rangle$ and $d_2 = \langle V_2, c, P_2\rangle$ that have the same constraint c and their combination $d = \langle V_1 \cup V_2, c, M'\rangle$, where $M' = \mathrm{met}((d_1\&d_2)|d_1|d_2)$. The combined dataflow constraint has a larger set of methods for solving the constraint c.

5. UI examples

Prior to fitting our constraint systems semantics to the institution settings, the topic of the next section, we relate the constructions thus far to practical programming. We have achieved the following: for an interface I with model μ we can flexibly combine independently developed dataflow constraints $d_1 = \langle V_1, c_1, M_1\rangle, \ldots, d_n = \langle V_n, c_n, M_n\rangle$.

First we need a common set of variables V and for each d_i a substitution $s_i : V_i \to E_{I,V}$. This aligns all constraints on the same set of variables, possibly with local transformations on each constraint system's methods and local constraints. If each transformation is sufficiently well behaved, i.e., they are mostly renamings on each d_i's output variables, the resulting constraint system will have enough methods to satisfy the desired constraints efficiently.

We demonstrate with the familiar image scaling GUI example how substitution and composing (with conjunction) of dataflow constraints enable reuse when constructing constraint systems.

Example 5.1. In our example GUI for scaling an image, see Section 2 and Example 3.18, the user can specify the width of the image either as the number of pixels w_a or as a scaling factor w_r to be applied to the initial width w_i of the image; and similarly for the corresponding height variables h_i, h_a, and h_r. The two dataflow constraints, one between w_a, w_i, and w_r, and the other between h_a, h_i, and h_r, can be constructed by substitution from the same dataflow constraint

$$G = \langle\{v_i : \mathbf{int}, v_a : \mathbf{int}, v_r : \mathbf{float}\}, \lfloor v_i v_r\rfloor == v_a, \{(v_a := \lfloor v_i v_r\rfloor), (v_r := v_a/v_i)\}\rangle.$$

We assume normal semantics for the types and operations and that $v_i \neq 0$. The substitutions $s_w = [v_i \mapsto w_i, v_a \mapsto w_a, v_r \mapsto w_r]$ and $s_h = [v_i \mapsto h_i, v_a \mapsto h_a, v_r \mapsto h_r]$ maps G to the global variables $w_i, w_a, w_r, h_i, h_a, h_r$ as desired. The conjunction of these dataflow constraints is a dataflow constraint whose constraint defines when an image scaling GUI is in a consistent state and whose methods provide the means to bring the GUI into such a state:

$$\begin{aligned} s_w(G)\ \&\ s_h(G) = \langle\{&w_i : \mathbf{int}, w_a : \mathbf{int}, w_r : \mathbf{float}, h_i : \mathbf{int}, h_a : \mathbf{int}, h_r : \mathbf{float}\},\\ &\lfloor w_i w_r\rfloor == w_a\ \&\ \lfloor h_i h_r\rfloor == h_a,\\ &\{(w_a, h_a := \lfloor w_i w_r\rfloor, \lfloor h_i h_r\rfloor),\\ &\ (w_a, h_r := \lfloor w_i w_r\rfloor, h_a/h_i), \end{aligned}$$

$$(w_r, h_a := w_a/w_i, \lfloor h_i h_r \rfloor),$$
$$(w_r, h_r := w_a/w_i, h_a/h_i)\}\rangle.$$

Here we see that the first method $(w_a, h_a := \lfloor w_i w_r \rfloor, \lfloor h_i h_r \rfloor)$ follows automatically from the constraint $\lfloor w_i w_r \rfloor == w_a$ & $\lfloor h_i h_r \rfloor == h_a$. Since the constraint is affine, we could conceivably use a computer algebra tool to also generate the remaining methods.

Assume now a GUI otherwise the same, except for requiring that the width and height are scaled by the same factor r. Define $r_r = [w_r \mapsto r, h_r \mapsto r]$ to map the width and height ratios to the common ratio r. Thus, the global variables are $V = \{w_i, w_a, h_i, h_a, r\}$. A suitable dataflow constraint for this GUI is attained by using slightly different substitutions. Let $r_w = \lceil r_r \rceil^V \circ s_w = [v_i \mapsto w_i, v_a \mapsto w_a, v_r \mapsto r]$ and $r_h = \lceil r_r \rceil^V \circ s_h = [v_i \mapsto h_i, v_a \mapsto h_a, v_r \mapsto r]$. Then

$$\begin{aligned} r_w(G)\,\&\,r_h(G) = \langle\{& w_i : \mathbf{int}, w_a : \mathbf{int}, h_i : \mathbf{int}, h_a : \mathbf{int}, r : \mathbf{float}\}, \\ & \lfloor w_i r \rfloor == w_a \,\&\, \lfloor h_i r \rfloor == h_a, \\ & \{(w_a, h_a := \lfloor w_i r \rfloor, \lfloor h_i r \rfloor), \\ & \;(w_a, r := \lfloor w_i (h_a/h_i) \rfloor, h_a/h_i), \\ & \;(h_a, r := \lfloor h_i (w_a/w_i) \rfloor, w_a/w_i)\}\rangle. \end{aligned}$$

The system obtained by $\lceil r_a \rceil^{\{w_i, w_a, h_i, h_a\}}(s_w(G)\ \&\ s_h(G))$ is much smaller,

$$\begin{aligned} &\langle\{w_i : \mathbf{int}, w_a : \mathbf{int}, h_i : \mathbf{int}, h_a : \mathbf{int}, r : \mathbf{float}\}, \\ &\;\lfloor w_i r \rfloor == w_a \,\&\, \lfloor h_i r \rfloor == h_a, \\ &\;\{(w_a, h_a := \lfloor w_i r \rfloor, \lfloor h_i r \rfloor)\}\rangle. \end{aligned}$$

All other methods fail the substitution safety criteria for applying $\lceil r_r \rceil^V$ to them.

We now give a second example on modelling and implementing a GUI as a dataflow constraint. The focus is on dynamic composition of GUI fragments and reasoning about global properties that should hold in such compositions.

Example 5.2. See Fig. 3 for a GUI for scheduling a conference day, where the day starts at time S and ends at time E, and each *agenda item* i has a start time s_i, duration d_i, and end time e_i. An agenda item must satisfy $e_i == s_i + d_i$; we assume the base API defines $+$ for adding a duration to time (and similarly $-$ for subtracting), and also overloads $+$ for summing two durations. A dataflow constraint for an agenda item would have at least a method that computes e from s and d, but depending on the desired GUI functionality, possibly also one that computes s from e and d. In this example we choose both, and the dataflow constraint for agenda items is

$$D_1 = \langle\{s : \text{date}, d : \text{duration}, e : \text{date}\}, e == s + d, \{e := s + d, s := e - d\}\rangle.$$

Each consecutive pair of agenda items must satisfy, $e_i == s_{i+1}$. Again, based on the desired functionality, whether earlier agenda items should be adjusted after changes on later ones, the dataflow constraint that connects two adjacent agenda items may have one or two methods. We again choose the design where data can flow to both directions:

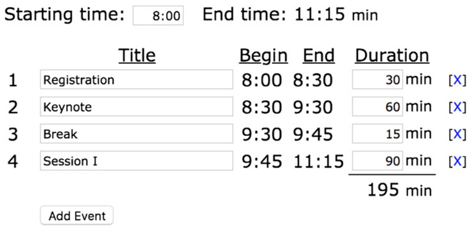
$$D_2 = \langle\{e_{\text{prev}}, s_{\text{next}}\}, e_{\text{prev}} == s_{\text{next}}, \{s_{\text{next}} := e_{\text{prev}}, s_{\text{prev}} := e_{\text{next}}\}\rangle.$$

Finally, the first (index 0) and last (index n) agenda items must satisfy $s_0 == S$ and $e_n == E$. Suitable substitutions on C_2 gives dataflow constraints that can model these dependencies.

Assuming the program variables S and E, as well as s_i, d_i, and e_i for $i = 0, \ldots, n$, are bound to GUI widgets appropriately, we get a well-behaved GUI by constructing the following new dataflow constraints via substitution:

$$\begin{aligned} K_S &= [e_{\text{prev}} \mapsto S, s_{\text{next}} \mapsto s_0](D_2), \\ K_E &= [e_{\text{prev}} \mapsto e_n, s_{\text{next}} \mapsto E](D_2), \\ A_i &= [s \mapsto s_i, d \mapsto d_i, e \mapsto e_i](D_1), \text{for all } i = 0, \ldots, n, \\ R_i &= [e_{\text{prev}} \mapsto e_i, s_{\text{next}} \mapsto s_{i+1}](D_2), \text{for all } i = 0, \ldots, n-1. \end{aligned}$$

Further, our semantics of dataflow constraints allows for easy reasoning about invariants that the conjunction of the constraints maintain. For example, a correct GUI always satisfies $S + d_0 + \cdots + d_n == E$. Each conjunction $L == A_i \& R_i$ gives the constraint $e_i == s_i + d_i$ & $s_{i+1} == e_i$, which simplifies to $s_{i+1} == s_i + d_i$. Easy inductive argument shows that in the conjunction $L_0 \& \cdots \& L_{n-1}$, $s_n == s_0 + \sum_{i=0,\ldots,n-1} d_i$ holds. The invariant then follows directly from the conjunction $K_S \& (L_0 \& \cdots \& L_{n-1}) \& A_n \& K_E$.



Starting time: 8:00 End time: 11:15 min

	Title	Begin	End	Duration	
1	Registration	8:00	8:30	30 min	[X]
2	Keynote	8:30	9:30	60 min	[X]
3	Break	9:30	9:45	15 min	[X]
4	Session I	9:45	11:15	90 min	[X]
				195 min	

Add Event

Fig. 3. A GUI implemented as a dynamic composition of fragments of GUI logic represented with dataflow constraints.

The above example is rather simple, but it highlights how fragments of GUI logic compose, and how the programmer can be assured that the composition is well-behaved. We implemented the example using our constraint system based GUI library HotDrink [24]; Fig. 3 shows a snapshot. Users can add, remove, and reorder agenda items at will, but as these operations are defined in terms of the above substitutions, we can be certain that the invariant is respected.

6. Institution based module system

An institution [13] is a framework for relating syntax, specifications and models. It can be interpreted as a framework for modular reuse of specification and code. We use this interpretation to formalise a module framework for constraint systems. The restrictions the institution concept places ensures that such a module system will be very well behaved, yet have significantly more powerful reuse mechanisms, namely signature morphisms, than standard module systems. In addition, the institution framework allows the use of institution-independent tools, such as the institution-independent specification structuring mechanisms [30] and syntactic theory functors for reuse and growing institutions piecemeal [31]. Institution theory is very general, phrased in terms of categories, functors and a satisfaction relation. We will give a short overview of the needed concepts.

Many authors have investigated relationships between institutions, e.g., reuse based on theoroidal comorphisms (formerly called maps of institutions) [32,33]. Most of this work is tied to logical frameworks, in line with Goguen and Burstall's original motivation for institutions to provide a framework for studying logical model-theory [13]. It is beyond the scope of this paper to cover these topics.

Multiway dataflow constraint systems are built by coordinating individual dataflow constraints on a set of global variables. A modular reuse system needs to place the individual dataflow constraints appropriately in the global systems. Central is the mapping of local variables from each dataflow constraint to the global setting.

In the next subsection we define the notions of categories and functors that we need. All the needed building blocks have already been presented. Then we define the notion of institution, and instantiate that as a modular framework for constraints and dataflow constraints.

6.1. Categories for constraint systems

As a reminder, a *category* consists of objects and morphisms. Morphisms connect two objects: the source and the target. Every object A has an identity morphism $\mathsf{id}_A : A \to A$. Every two morphisms $f : A \to B$ and $g : B \to C$ can be composed to a morphism $f; g : A \to C$. The identity morphism is the identity for composition, i.e., for $f : A \to B$, we have that $\mathsf{id}_A; f = f$ and $f; \mathsf{id}_B = f$. Composition is associative, i.e., for $f : A \to B$, $g : B \to C$ and $h : C \to D$, $(f; g); h = f; (g; h)$. The canonical example of categories is **Set** that has sets as objects, total functions between sets as morphisms, identity functions on sets as identity morphisms, and function composition as composition of morphisms.

Definition 6.1. For an interface I, model $\mu : I \to \mathbf{Set}$ and variables V for I, let $A_{\mu,V}$ be the set of allocations from V to μ. The discrete category $\mathbf{All}_{I,V,\mu}$ is defined by

- objects: the powerset of $A_{\mu,V}$, i.e., every subset $A \subseteq A_{\mu,V}$ is an object,
- morphisms: a single morphism $\mathsf{id}_A : A \to A$ for every $A \subseteq A_{\mu,V}$,
- identity morphism: the single morphism $\mathsf{id}_A : A \to A$ has to be the identity morphism for the object $A \subseteq A_{\mu,V}$,
- composition: for every morphism $\mathsf{id}_A : A \to A$, the composition $\mathsf{id}_A; \mathsf{id}_A = \mathsf{id}_A$ (the only possible choice).

Definition 6.2. For an interface I, the category $\mathbf{Subvar}_I$ is defined by

- objects: an object for every collection of variables V,
- morphisms from object V to V': all substitutions $s : V \to E_{I,V'}$ from V to expressions on V',
- identity morphism on V: the identity substitution on variables V, and
- composition: composition of substitutions.

Here we see that a morphism does not have to be a function between two objects. The substitutions take us from variables to expressions, but have an associative composition operation with the identity substitution as the neutral element, as required for a category.

We may construct categories with different objects but from the same morphisms as above.

Definition 6.3. For a staid interface I, the category $\mathbf{Con}_I$ is defined by

- objects: an object is $C_{I,V}$, the set of all constraints, for every collection of variables V,
- morphisms from object $C_{I,V}$ to $C_{I,V'}$: all substitutions $s : V \to E_{I,V'}$ interpreted as substitutions on constraints,
- identity morphism on $C_{I,V}$: the identity substitution on variables V, and
- composition: composition of substitutions.

Given an element $c \in C_{I,V}$, the substitution maps it to an element $s(c) \in C_{I,V'}$. In this case we have that the morphisms actually are functions from constraints to constraints, with function composition as morphism composition. Thus $\mathbf{Con}_I$ is a subcategory of $\mathbf{Set}$, written as $\mathbf{Con}_I \subseteq \mathbf{Set}$.

Definition 6.4. For a staid interface I, the category $\mathbf{DFC}_I$ is defined by

- objects: an object is $D_{I,V}$, the set of all dataflow constraints, for every collection of variables V,
- morphisms from object $D_{I,V}$ to $D_{I,V'}$: all substitutions $s : V \to E_{I,V'}$ interpreted as substitutions on dataflow constraints,
- identity morphism on $D_{I,V}$: the identity substitution on variables V, and
- composition: composition of substitutions.

Recall the monoid action properties on dataflow constraints (Proposition 4.18) to see that this actually is a category: calling the composition of two substitutions or calling them indvidually in sequence give the same result $(s_2 \circ s_1)(d) = s_2(s_1(d))$ as required. Here we also map sets to sets using the substitution function, thus $\mathbf{DFC}_I \subseteq \mathbf{Set}$.

Another twist on making categories is the dual (or opposite) category. A dual category has the same objects and morphisms as the original category, but the morphisms have swapped source and target nodes (so the direction of composition also changes). Recall that methods behave dually to substitutions, with composition in the opposite order. Thus reversing the arrows of the above categories is replacing every substitution $s : V \to E_{I,V'}$ by the method $s^{\ominus} : V \leftarrow E_{I,V'}$, and we achieve dual categories. In general, dual categories can be constructed even if the reversed morphisms do not have reasonable interpretations. We can always restore the original category by taking the dual of the dual category; in the example case this is switching back from methods to substitutions.

6.2. Functors

A functor is a mapping between categories. Given two categories $\mathbf{C}$ and $\mathbf{D}$, a *functor* $F : \mathbf{C} \to \mathbf{D}$ maps objects to objects and morphisms to morphisms, preserving identity morphisms and composition: for every object A in $\mathbf{C}$, $F(\mathsf{id}_A) = \mathsf{id}_{F(A)}$, and for every morphism $f : A \to B$ and $g : B \to C$ in $\mathbf{C}$, $F(f;g) = F(f);F(g)$.

For example, we have a functor $\mathrm{con}_I : \mathbf{Subvar}_I \to \mathbf{Con}_I$, given by $\mathrm{con}_I(V) = C_{I,V}$, which maps a collection of variables to the constraints on those variables, and $\mathrm{con}_I(s : V \to V') = s : C_{I,V} \to C_{I,V'}$, which maps the substitution on variables to the substitution on the constraints. Likewise we have a functor $\mathrm{dfc}_I : \mathbf{Subvar}_I \to \mathbf{DFC}_I$ which takes variables V for I to dataflow constraints $D_{I,V}$, and maps substitutions in $\mathbf{Subvar}_I$ to substitutions in $\mathbf{DFC}_I$. Interestingly, we also have functor $\mathrm{dcn}_I : \mathbf{DFC}_I \to \mathbf{Con}_I$ that extracts the constraint $\mathsf{con}(d)$ from every dataflow constraint $d \in D_{I,V}$. Such a functor is called forgetful, since it forgets some of the structure of the source when mapping to the target.

We can define a functor $\mathrm{all}_s : \mathbf{All}_{I,V',\mu} \to \mathbf{All}_{I,V,\mu}$ for a substitution $s : V \to E_{I,V'}$ on variables.

- For an object A' in $\mathbf{All}_{I,V',\mu}$, i.e., $A' \subseteq A_{\mu,V'}$, define the set of allocations $\mathrm{all}_s(A') = \{[\![s]\!]_{\mu,\_}(a') \mid a' \in A'\} \in A_{\mu,V}$.
- An identity mapping $id_{A'}$ is mapped to the identity mapping $id_{\mathrm{all}_s(A')}$ by $\mathrm{all}_s(id_{A'})$, and
- composition in $\mathbf{All}_{I,V',\mu}$ is mapped to composition in $\mathbf{All}_{I,V,\mu}$ by all_s, basically since there is no other choice.

Now functors compose by the composition of mapping objects to objects and morphisms to morphisms, and the combined mapping obeys the functor requirements. This also admits an identity functor, mapping a category to itself, by mapping each object and each morphism to itself. It is now possible to define a category where the objects are the categories, the morphisms are the functors, morphism composition is functor compostition, and the identity morphism is the identity functor. This category is referred to as $\mathbf{CAT}$.

Definition 6.5. For an interface I with model $\mu : I \to \mathbf{Set}$, define the category $\mathbf{All}_{I,\mu}$ by

- objects: the categories $\mathbf{All}_{I,V,\mu}$ for every collection of variables V for I,

- morphisms from object $\mathbf{All}_{I,V',\mu}$ to $\mathbf{All}_{I,V,\mu}$: the functors $\mathrm{all}_s : \mathbf{All}_{I,V',\mu} \to \mathbf{All}_{I,V,\mu}$ for every substitution $s : V \to E_{I,V'}$,
- identity morphism on $\mathbf{All}_{I,V',\mu}$: the identity functor on $\mathbf{All}_{I,V',\mu}$,
- composition: the composition of functors $\mathrm{all}_{s'} : \mathbf{All}_{I,V'',\mu} \to \mathbf{All}_{I,V',\mu}$ with $\mathrm{all}_s : \mathbf{All}_{I,V',\mu} \to \mathbf{All}_{I,V,\mu}$ is the functor $\mathrm{all}_{s;s'} : \mathbf{All}_{I,V'',\mu} \to \mathbf{All}_{I,V,\mu}$.

We see that $\mathbf{All}_{I,\mu} \subseteq \mathbf{CAT}$.

The astute reader will notice that we can define a functor $\mathrm{all}_{I,\mu} : \mathbf{Subvar}_I^{\mathrm{op}} \to \mathbf{All}_{I,\mu}$ by $\mathrm{all}_{I,\mu}(V) = \mathbf{All}_{I,V,\mu}$ and $\mathrm{all}_{I,\mu}(s^{\ominus} : V \leftarrow E_{I,V'}) = \mathrm{all}_s : \mathbf{All}_{I,V',\mu} \to \mathbf{All}_{I,V,\mu}$. It goes from the dual of the category of variables and substitutions to the category of categories of allocations. A functor from the dual category is called a *contravariant functor* to draw attention to this trait.

6.3. Institutions for constraints

Definition 6.6. An institution $\mathcal{INST}$ consists of

- a category **Sig** of signatures,
- a functor $\mathrm{spec} : \mathbf{Sig} \to \mathbf{Set}$ of specifications
- a contravariant functor $\mathrm{mod} : \mathbf{Sig}^{\mathrm{op}} \to \mathbf{CAT}$ of models,
- a satisfaction relation $\models$ which for every object A of **Sig** defines a relation $\models_A \subseteq \mathrm{mod}(A) \times \mathrm{spec}(A)$,

such that the satisfaction condition holds: for every morphism $f : A \to A'$ in **Sig**,

$$\mu' \models_{A'} \big(\mathrm{spec}(f : A \to A')\big)(c) \iff \big(\mathrm{mod}(f^{\mathrm{op}} : A' \to A)\big)(\mu') \models_A c,$$

where μ' is an object of $\mathrm{mod}(A')$ (model), and $c \in \mathrm{spec}(A)$ (specification).

We can now put together our categories and functors and define the institution of constraints. To ease reading of quantified formulas, we use $\cdot$ as a separator between the declaration of the quantified set X and the predicate p, thus we consider $\forall x \in X \cdot p$ as being clearer than $\forall x : Xp$ or $\forall (x : X) p$ especially when $x : X$ is a large expression.

Definition 6.7. The institution $\mathcal{CONS}_{I,\mu}$ of constraints and allocations for a staid interface I with standard model μ consists of

- the category $\mathbf{Subvar}_I$ of variables and substitutions,
- the functor $\mathrm{con}_I : \mathbf{Subvar}_I \to \mathbf{Con}_I$ of constraints and substitutions,
- the contravariant functor $\mathrm{all}_{I,\mu} : \mathbf{Subvar}_I^{\mathrm{op}} \to \mathbf{All}_{I,\mu}$ of allocations from variables to μ,
- the V-indexed satisfaction relations $\models_{I,\mu,V} \subseteq \mathrm{all}_{I,\mu}(V) \times \mathrm{con}_I(V)$ for variables V as an object of $\mathbf{Subvar}_I$, defined by

  $$A \models_{I,\mu,V} c \iff \forall (a : V \to \mu) \in A \cdot [\![c]\!]_{\mu,a}$$

  for allocations $a : V \to \mu$ in object A of $\mathbf{All}_{I,V,\mu} = \mathrm{all}_{I,\mu}(V)$ and constraint $c \in C_{I,V} = \mathrm{con}_I(V)$.

Theorem 6.8. *The claim that $\mathcal{CONS}_{I,\mu}$ is an institution holds.*

Proof. The category $\mathbf{Con}_I \subseteq \mathbf{Set}$ and the category $\mathbf{All}_{I,\mu} \subseteq \mathbf{CAT}$, ensuring the functors have correct codomains. For a substitution $s : V \to E_{I,V'}$, we must show that for every object $A' \in \mathbf{All}_{I,V',\mu} = \mathrm{all}_{I,\mu}(V')$, i.e., $A' \subseteq A_{I,V'}$, and every constraint $c \in C_{I,V} = \mathrm{con}_I(V)$

$$A' \models_{I,\mu,V'} \big(\mathrm{con}_I(s : V \to E_{I,V'})\big)(c) \iff \big(\mathrm{all}_{I,\mu}(s^{\ominus} : E_{I,V'} \to V)\big)(A') \models_{I,\mu,V} c,$$

which becomes, by expanding the definitions on each side,

$$\begin{aligned} A' \models_{I,\mu,V'} \big(\mathrm{con}_I(s : V \to E_{I,V'})\big)(c) &\iff \forall (a' : V' \to \mu) \in A' \cdot [\![s(c)]\!]_{\mu,a'}, \\ \big(\mathrm{all}_{I,\mu}(s^{\ominus} : E_{I,V'} \to V)\big)(A') \models_{I,\mu,V} c &\iff \forall (a : V \to \mu) \in \{[\![s]\!]_{\mu,a'} \mid a' \in A'\} \cdot [\![c]\!]_{\mu,a} \\ &\iff \forall (a' : V' \to \mu) \in A' \cdot [\![c]\!]_{\mu,[\![s]\!]_{\mu,a'}}, \end{aligned}$$

respectively. Since $[\![s(c)]\!]_{\mu,a'} = [\![c]\!]_{\mu,[\![s]\!]_{\mu,a'}}$ the claim holds. □

Definition 6.9. The institution $\mathcal{DFC}_{I,\mu}$ of dataflow constraints and allocations for a staid interface I with standard model μ consists of

- the category $\mathbf{Subvar}_I$ of variables and substitutions,

- the functor $\mathrm{dfc}_I : \mathbf{Subvar}_I \to \mathbf{DFC}_I$ of dataflow constraints and substitutions,
- the contravariant functor $\mathrm{all}_{I,\mu} : \mathbf{Subvar}_I^{\mathrm{op}} \to \mathbf{All}_{I,\mu}$ of allocations from variables to μ,
- the V-indexed satisfaction relations $\models_{I,\mu,V} \subseteq \mathrm{all}_{I,\mu}(V) \times \mathrm{dfc}_I(V)$ for variables V as an object of $\mathbf{Subvar}_I$, defined by

$$A \models_{I,\mu,V} d \iff \forall(a : V \to \mu) \in A, m \in \mathsf{met}(d) \cdot [\![\mathsf{con}(d)]\!]_{\mu,[\![m]\!]_{\mu,a}}$$

for all allocations $a : V \to \mu$ in object A of $\mathbf{All}_{I,V,\mu} = \mathrm{all}_{I,\mu}(V)$ and constraint $d \in D_{I,V}$ where $D_{I,V} = \mathrm{dfc}_I(V)$.

Theorem 6.10. *The claim that $\mathcal{DFC}_{I,\mu}$ is an institution holds.*

Proof. The category $\mathbf{DFC}_I \subseteq \mathbf{Set}$ and the category $\mathbf{All}_{I,\mu} \subseteq \mathbf{CAT}$, ensuring the functors have correct codomains. For a substitution $s : V \to E_{I,V'}$, we must show that for every object $A' \in \mathbf{All}_{I,V',\mu} = \mathrm{all}_{I,\mu}(V')$, i.e., $A' \subseteq A_{I,V'}$, and every dataflow constraint $d \in D_{I,V} = \mathrm{dfc}_I(V)$

$$A' \models_{I,\mu,V'} \left(\mathrm{dfc}_I(s : V \to E_{I,V'})\right)(d) \iff \left(\mathrm{all}_{I,\mu}(s^{\ominus} : E_{I,V'} \to V)\right)(A') \models_{I,\mu,V} d,$$

which becomes, by expanding the definitions on each side,

$$\begin{aligned}
A' \models_{I,\mu,V'} \left(\mathrm{dfc}_I(s : V \to E_{I,V'})\right)(d) &\iff \forall(a' : V' \to \mu) \in A', m' \in \mathsf{met}(s(d)) \cdot [\![\mathsf{con}(s(d))]\!]_{\mu,[\![m']\!]_{\mu,a'}} \\
&\iff \forall(a' : V' \to \mu) \in A', m' \in \mathsf{met}(s(d)) \cdot [\![m'^{\Phi V} s(\mathsf{con}(d))]\!]_{\mu,a'}, \\
\left(\mathrm{all}_{I,\mu}(s^{\ominus} : E_{I,V'} \to V)\right)(A') \models_{I,\mu,V} d &\iff \forall(a : V \to \mu) \in \{[\![s]\!]_{\mu,a'} \mid a' \in A'\}, m \in \mathsf{met}(d) \cdot [\![con(d)]\!]_{\mu,[\![m]\!]_{\mu,a}} \\
&\iff \forall(a : V \to \mu) \in \{[\![s]\!]_{\mu,a'} \mid a' \in A'\}, m \in \mathsf{met}(d) \cdot [\![m^{\Phi V} con(d)]\!]_{\mu,a} \\
&\iff \forall(a' : V' \to \mu) \in A', m' \in \mathsf{met}(s(d)) \cdot [\![m'^{\Phi V} s(\mathsf{con}(d))]\!]_{\mu,a'},
\end{aligned}$$

respectively. Thus the claim holds. □

7. Multiway dataflow constraint systems

A *multiway dataflow constraint system*, *constraint system* for short, is a collection D of dataflow constraints and a goal. The methods of each dataflow constraint $d \in D$ locally establish that dataflow constraint's constraint. Together the conjunction of the contraints of D should establish the constraint system's goal. Thus an approriate sequence of one dataflow method from each $d \in D$ will establish the goal whenever one of the system's variables is modified. The purpose is to let a planner select the local methods from each dataflow constraint $d \in D$ in order to achieve the global goal.

Definition 7.1. Let I be a staid interface with some staid semantics.

A *constraint system* is a triple $\langle V, g, D\rangle$ where V are variables for I, $g \in C_{I,V}$ is a constraint called the constraint system's *goal*, and $D \subseteq D_{I,V}$ is a set of dataflow constraints.

The *meaning* of a constraint system $[\![\langle V, g, D\rangle]\!] = \&_{d\in D} d$, the dataflow constraint being the conjunction of all dataflow constraints in D.

The global variables V for a constraint system coordinate all its parts. Each of the dataflow constraints handle a local set of variables, related to the global set by an inclusion. The meaning of the constraint system is a syntactic translation to a system wide (global) dataflow constraint. The semantics of the global dataflow constraint is the semantics of the system.

A planner selects methods from $\mathsf{met}(\cup_{d\in D} d)$ repeatedly in order to achieve the goal g. To facilitate planning, a planner algorithm imposes its own wellformedness requirements on the contained dataflow constraints. One requirement might be that if two dataflow constraints $d, d' \in D$ have the same set of variables $\mathsf{var}(d) = \mathsf{var}(d')$ then $d = d'$, and that for every $d \in D$ whenever two methods $m, m' \in \mathsf{met}(d)$ have the same set of output variables $\mathsf{out}(m) = \mathsf{out}(m')$ then $m = m'$. The so called *method restriction* [34, p. 56] requires even that there are no two methods $m, m' \in \mathsf{met}(d)$ such that $\mathsf{out}(m) \subseteq \mathsf{out}(m')$. Assuming method restriction, there are planners with polynomial worst-case time complexity [35]. These syntactically detectable restrictions ensure that a planner will always terminate, even if the sets of methods contain other errors or are not compatible with their stated constraint. In fact most work on planners omit explicit constraints and just focus on the set of methods $\mathsf{met}(\cup_{d\in D} d)$, thus ignoring the ability to verify the constraint system against an explicitly stated goal.

Definition 7.2. Let I be a staid interface and μ a standard model for I.

A constraint system $\langle V, g, D\rangle$ is *sound* in μ when

1. each $d \in D \subseteq D_{I,V,\mu}$ is sound in μ, i.e., $\mathsf{met}(D) \models_{I,\mu,V} \mathsf{con}(d)$, and
2. $\mathsf{con}(\&_{d\in D} d) \models_{I,V,\mu} g$.

For a sound constraint system the participating dataflow constraints each contribute towards the goal. A sound constraint system $\langle V, g, D\rangle$ is *comprehensive* in μ when for all allocations $a \in A_{\mu,V}$ there exists a method $m \in \mathsf{met}(\&_{d\in D} d)$ such that $\llbracket g\rrbracket_{\mu,\llbracket m\rrbracket_{\mu,a}}$ holds. Ideally a planner's wellformedness requirements should imply that the constraint system is comprehensive. If the constraint system is not comprehensive, the planner may be unable to select any method from $\mathsf{met}(\cup_{d\in D} d)$ to achieve progress. This may happen if the dataflow constraint combinators end up with an empty set of methods. Also in this situation the planner will terminate, but without meeting its goal. We will not discuss this further here.

We present three reuse mechanisms for constraint systems: substitution for adapting to a different set of global variables, and conjunction and disjunction for combining constraint systems. We also have two specific constraint systems.

Definition 7.3. The constraint system true $\mathrm{CST} = \langle \emptyset, \mathrm{TRUE}, \{T\}\rangle$ based on the dataflow constraint T.

The constraint system false $\mathrm{CSF} = \langle \emptyset, \mathrm{FALSE}, \{\}\rangle$.

These two constraint systems are both sound and comprehensive for any staid interface since the set of variables is empty.

Definition 7.4. Let I be a staid interface and V, W be variables for I.

Reuse of a constraint system $C = \langle V, g, D\rangle$ by a substitution $s : V \to E_{I,W}$ is the triple $s(C) = \langle W, s(g), s(D)\rangle$.

It is easy to see that the reuse of a constraint system by a substitution yields a constraint system.

Proposition 7.5. *Let I be a staid interface, μ be a standard model for I and $s : V \to E_{I,W}$ be a substitution.*

Substitution on constraint systems in staid semantics preserve meaning, i.e., $s(\llbracket C\rrbracket) \subseteq \llbracket s(C)\rrbracket$.

If the constraint system $C = \langle V, g, D\rangle$ is sound for μ, then $s(C)$ is also sound for μ.

Proof. Meaning is preserved since $s(\llbracket C\rrbracket) = s(\llbracket\langle V, g, D\rangle\rrbracket) = s(\&_{d\in D} d) \subseteq \&_{d\in s(D)} d = \llbracket\langle W, s(g), s(D)\rangle\rrbracket = \llbracket s(C)\rrbracket$.

Preservation of soundness follows from the satisfaction condition for the institutions $\mathcal{DFC}_{I,\mu}$ and $\mathcal{CONS}_{I,\mu}$. The former takes sound dataflow constraints to sound dataflow constraints using the substitution. The latter matches the conjunction of constraints from the dataflow constraint with the goal before and after the substitution, i.e., $s(\mathsf{con}(\&_{d\in D} d)) \models_{I,V,\mu} s(g)$ since C is sound. □

It then follows that substitutions are a monoid action for the reuse of constraint systems. The monoid action also preserves soundness of the constraint systems. Comprehensiveness may be destroyed by this reuse, since, depending on the specific substitution, some methods in $d \in D$ may have disappeared.

Definition 7.6 *(Conjunction of constraint systems).* Let I be a staid interface with staid semantics.

The conjunction of two constraint systems $C_1 = \langle V, g_1, D_1\rangle$ and $C_2 = \langle V, g_2, D_2\rangle$ is the triple $C_1\&C_2 = \langle V, g_1\&g_2, D_1\&D_2\rangle$ where $D_1\&D_2 = \{d_1\&d_2 \mid d_1 \in D_1, d_2 \in D_2\}$.

It is easy to see that the conjunction of two constraint systems yields a constraint system: the conjunction of two dataflow constraints is a dataflow constraint, and the conjunction of two constraints is a constraint. The two constraint systems C_1 and C_2 are on the same set of variables V. If this is not the case, then each of them can be aligned by substitution from any local set of variables V_1 and V_2, respectively, to a global set of variables V before the conjunction.

Proposition 7.7. *Let I be s staid interface and μ a standard model for I, and $C_1 = \langle V, g_1, D_1\rangle$ and $C_2 = \langle V, g_2, D_2\rangle$ be constraint systems.*

Conjunction on constraint systems with staid semantics preserve meaning, i.e., $\llbracket C_1\&C_2\rrbracket = \llbracket C_1\rrbracket\&\llbracket C_2\rrbracket$.

If the two constraint systems C_1 and C_2 are sound for μ, then $C_1\&C_2$ is also sound for μ.

Proof. Meaning is preserved since $(\&_{d_1\in D_1} d_1)\&(\&_{d_2\in D_2} d_2) = \&_{d\in\{d_1\&d_2 \mid d_1\in D_1, d_2\in D_2\}} d = D_1\&D_2$.

The conjunction of sound dataflow constraints on compatible variables is a sound dataflow constraint, and since each of the individual constraint systems is sound with respect to its individual goal, the conjunction of the goals is implied by $\mathsf{con}(\&_{d\in D_1\&D_2} d)$. □

It now follows that conjunction of constraint systems is associative and commutative, with CST as unit and CSF as zero. Comprehensiveness may be lost due to the interaction between methods when combining them.

Definition 7.8 *(Disjunction of constraint systems).* Let I be a staid interface with staid semantics.

The disjunction of two constraint systems $C_1 = \langle V, g_1, D_1\rangle$ and $C_2 = \langle V, g_2, D_2\rangle$ is $C_1 \mid C_2 = \langle V, g_1|g_2, D\rangle$, where $D = \big((\&_{d\in D_1\setminus D_2} d) \mid (\&_{d\in D_2\setminus D_1} d)\big)\&(\&_{d\in D_1\cap D_2} d)$.

It is easy to see that the disjunction of two constraint systems yields a constraint system since all combinators yield the appropriate results. The two constraint systems C_1 and C_2 may be aligned to the same set of global variables V if necessary for the disjunction.

Proposition 7.9. *Let I be a staid interface and μ a standard model for I, and $C_1 = \langle V, g_1, D_1\rangle$ and $C_2 = \langle V, g_2, D_2\rangle$ be constraint systems.*

Disjunction on constraint systems with staid semantics preserve meaning, i.e., $[\![C_1 \mid C_2]\!] = [\![C_1]\!] \mid [\![C_2]\!]$.

If the two constraint systems C_1 and C_2 are sound for μ, then $C_1 \mid C_2$ is also sound for μ.

Proof. Let $C_1 \mid C_2 = \langle V, g_1|g_2, D\rangle$. Preservation of meaning follows by expanding the definition D, and then using distributivity of conjunction over disjunction for dataflow constraints.

$$[\![C_1 \mid C_2]\!] = \&_{d\in D} d = \big((\&_{d\in D_1\setminus D_2} d) \mid (\&_{d\in D_2\setminus D_1} d)\big)\&(\&_{d\in D_1\cap D_2} d)$$
$$= \big((\&_{d\in D_1\setminus D_2} d)\&(\&_{d\in D_1\cap D_2} d)\big) \mid \big((\&_{d\in D_2\setminus D_1} d)\&(\&_{d\in D_1\cap D_2} d)\big) = \big((\&_{d\in D_1} d)\&(\&_{d\in D_2} d) = [\![C_1]\!] \mid [\![C_2]\!]\big),$$

Then we need to show that $\mathsf{con}(|_{d\in D}\, d) \models_{I,V,\mu} g_1|g_2$. Since

$$\mathsf{con}((\&_{d\in D_1\setminus D_2} d)\&(\&_{d\in D_1\cap D_2} d)) \models_{I,V,\mu} c_1 \text{and}$$
$$\mathsf{con}((\&_{d\in D_2\setminus D_1} d)\&(\&_{d\in D_1\cap D_2} d)) \models_{I,V,\mu} c_2$$

the claim follows.

The meaning for the disjunction of dataflow constraints D maintains soundness from each of the components. □

It now follows that disjunction of constraint systems is associative and commutative with CSF as the unit, because $\big((\&_{d\in D_1\setminus\{\}} d) \mid (\&_{d\in\{\}\setminus D_1} d)\big)\&(\&_{d\in D_1\cap\{\}} d) = \big((\&_{d\in D_1} d)\big)\big)\&(\&_{d\in\emptyset}) = D_1$. We cannot get idempotency of disjunction since the conjunction combinator on the dataflow constraints is not idempotent. Comprehensiveness may be lost due to the interaction between methods when combining them.

Example 7.10. This is a constraint system version of Example 5.1. When building the constraint system we first define the global set of variables and the goal,

$$V = \{w_i : \textbf{int}, w_a : \textbf{int}, w_r : \textbf{float}, h_i : \textbf{int}, h_a : \textbf{int}, h_r : \textbf{float}\},$$
$$g = \lfloor w_i w_r \rfloor == w_a \;\&\; \lfloor h_i h_r \rfloor == h_a,$$

then we find the relevant dataflow constraints and their mapping to the global set of variables, i.e., the dataflow constraint G and the substitutions s_w and s_h as in the example. The resulting constraint system is

$$\langle V, g, \{s_w(G), s_h(G)\}\rangle.$$

If we want to coordinate the aspect ratios for the width and the height, we can construct the constraint system $\langle\{w_i : \textbf{int}, w_a : \textbf{int}, r : \textbf{float}, h_i : \textbf{int}, h_a : \textbf{int}\},\ \lfloor w_i r\rfloor == w_a \;\&\; \lfloor h_i r\rfloor == h_a, \{r_w(G), r_h(G)\}\rangle$.

8. Summary and conclusion

In investigating multiway dataflow constraint systems as a programming language, we designed a module system with variable substitution (including renaming and name matching) as the basic reuse mechanism with conjunction and disjunction as combinators. We showed how to integrate a specification language with the programming language. In the presented system, neither the specification nor the programming language needs to be nailed down: what are the languages' built-in types and operations is defined by a base API, a fixed but arbitrary signature.

We fitted the formal model of dataflow constraints into the institution framework, using variables and substitutions as the signature category. This gave us both a solid footing for designing the module system, and a clear guidance for using global variables as the coordinating "signature" with substitution as a powerful reuse mechanism.

The specification and programming languages both operate in the same semantical domain, here on simple sets and set theoretic functions for the base signature. This semantic compatibility allows proving and testing the relationship between code (the constraint satisfaction methods of data flow constraints) and specifications (predicate expressions that define when dataflow constraints are satisfied). The presented approach thus establishes a firm validation approach for dataflow constraint systems.

The presented work is largely motivated by the widely recognised problem of programming user interfaces: the inability to reuse code that defines the behaviour of user interfaces. We propose multiway dataflow constraint systems as the foundation for programming GUIs, in lieu of event handling programming. The examples in this paper show that the presented semantics can model concrete multiway dataflow constraint systems that arise in practical graphical user interfaces, and guide their design and implementation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] I.E. Sutherland, Sketchpad: a man-machine graphical communication system, in: Proceedings of the May 21-23, 1963, Spring Joint Computer Conference, AFIPS'63 (Spring), ACM, New York, NY, USA, 1963, pp. 329–346.

[2] B.A. Myers, R.G. McDaniel, R.C. Miller, A.S. Ferrency, A. Faulring, B.D. Kyle, A. Mickish, A. Klimovitski, P. Doane, The Amulet environment: new models for effective user interface software development, IEEE Trans. Softw. Eng. 23 (6) (1997) 347–365, https://doi.org/10.1109/32.601073.

[3] B. Myers, D. Giuse, R. Dannenberg, B. Zanden, D. Kosbie, E. Pervin, A. Mickish, P. Marchal, Garnet: comprehensive support for graphical, highly interactive user interfaces, Computer 23 (11) (Nov. 1990) 71–85, https://doi.org/10.1109/2.60882.

[4] B. Vander Zanden, An incremental algorithm for satisfying hierarchies of multiway dataflow constraints, ACM Trans. Program. Lang. Syst. 18 (1) (Jan. 1996) 30–72, https://doi.org/10.1145/225540.225543.

[5] J. Järvi, G. Foust, M. Haveraaen, Specializing planners for hierarchical multi-way dataflow constraint systems, in: Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences, GPCE 2014, ACM, New York, NY, USA, 2014, pp. 1–10.

[6] J. Järvi, M. Haveraaen, J. Freeman, M. Marcus, Expressing multi-way data-flow constraint systems as a commutative monoid makes many of their properties obvious, in: Proceedings of the 8th ACM SIGPLAN Workshop on Generic Programming, WGP'12, ACM, New York, NY, USA, 2012, pp. 25–32.

[7] M. Sannella, Skyblue: a multi-way local propagation constraint solver for user interface construction, in: Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology, UIST'94, ACM, New York, NY, USA, 1994, pp. 137–146.

[8] T.P. McCartney, User interface applications of a multi-way constraint solver, Tech. Rep. WUCS-95-22, Washington University of Saint Louis, MO, 1995, Computer Science, Oct. 1995.

[9] S. Oney, B. Myers, J. Brandt, ConstraintJS: programming interactive behaviors for the web by integrating constraints and states, in: Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology, UIST'12, ACM, New York, NY, USA, 2012, pp. 229–238.

[10] K. Lin, D. Chen, G. Dromey, C. Sun, Maintaining constraints expressed as formulas in collaborative systems, in: 2007 International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2007), 2007, pp. 318–327.

[11] C. Demetrescu, I. Finocchi, A. Ribichini, Reactive imperative programming with dataflow constraints, in: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'11, ACM, New York, NY, USA, 2011, pp. 407–426.

[12] T. Felgentreff, A. Borning, R. Hirschfeld, Specifying and solving constraints on object behavior, J. Object Technol. 13 (4) (2014), https://doi.org/10.5381/jot.2014.13.4.a1 1–38.

[13] J.A. Goguen, R.M. Burstall, Institutions: abstract model theory for specification and programming, J. ACM 39 (1) (1992) 95–146, https://doi.org/10.1145/147508.147524.

[14] D. Sannella, A. Tarlecki, Foundations of Algebraic Specification and Formal Software Development, Software Development, Monographs in Theoretical Computer Science. An EATCS Series, Springer, 2012.

[15] P.D. Mosses, CASL Reference Manual, The Complete Documentation of the Common Algebraic Specification Language, Lecture Notes in Computer Science, vol. 2960, Springer, 2004.

[16] T. Mossakowski, C. Maeder, K. Lüttich, The heterogeneous tool set, Hets, in: O. Grumberg, M. Huth (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, Proceedings, in: Lecture Notes in Computer Science, vol. 4424, Springer, 2007, pp. 519–522.

[17] J.A. Goguen, Memories of ADJ, in: G. Rozenberg, A. Salomaa (Eds.), Current Trends in Theoretical Computer Science - Essays and Tutorials, in: World Scientific Series in Computer Science, vol. 40, World Scientific, 1993, pp. 76–81.

[18] J. Goguen, J. Thatcher, E. Wagner, An initial algebra approach to the specification, correctness and implementation of abstract data types, in: R. Yeh (Ed.), Current Trends in Programming Methodology, vol. 4, Prentice Hall, 1978, pp. 80–149.

[19] J.V. Guttag, J.J. Horning, The algebraic specification of abstract data types, Acta Inform. 10 (1978) 27–52, https://doi.org/10.1007/BF00260922.

[20] J.V. Guttag, J.J. Horning, S.J. Garland, K.D. Jones, A. Modet, J.M. Wing, Larch: Languages and Tools for Formal Specification, Texts and Monographs in Computer Science, Springer, 1993, p. 1993.

[21] H. Ehrig, B. Mahr, Fundamentals of Algebraic Specification 1: Equations and Initial Semantics, EATCS Monographs on Theoretical Computer Science, vol. 6, Springer, 1985, p. 1985.

[22] H. Ehrig, B. Mahr, Fundamentals of Algebraic Specification 2, EATCS Monographs on Theoretical Computer Science, vol. 21, Springer, 1990, p. 1990.

[23] J. Järvi, M. Marcus, S. Parent, J. Freeman, J.N. Smith, Algorithms for user interfaces, in: GPCE'09: Proc. of 8th International Conference on Generative Programming and Component Engineering, ACM, New York, NY, USA, 2009, pp. 147–156.

[24] G. Foust, J. Järvi, S. Parent, Generating reactive programs for graphical user interfaces from multi-way dataflow constraint systems, in: Proceedings of the 2015 International Conference on Generative Programming: Concepts and Experiences, GPCE 2015, ACM, New York, NY, USA, 2015, pp. 121–130.

[25] J. Freeman, J. Järvi, W. Kim, M. Marcus, S. Parent, Helping programmers help users, in: GPCE'11: Proc. of 10th International Conference on Generative Programming and Component Engineering, ACM, New York, NY, USA, 2011, pp. 177–184.

[26] L. Allison, Programming denotational semantics II, Comput. J. 28 (5) (1985) 480–486, https://doi.org/10.1093/comjnl/28.5.480.

[27] A.H. Bagge, V. David, M. Haveraaen, Testing with axioms in C++ 2011, J. Object Technol. 10 (10) (2011) 1–32, https://doi.org/10.5381/jot.2011.10.1.a10.

[28] R. Hamlet, Random testing, in: J. Marciniak (Ed.), Encyclopedia of Software Engineering, Wiley, 1994, pp. 970–978.

[29] J.-P. Bernardy, P. Jansson, K. Claessen, Testing polymorphic properties, in: A. Gordon (Ed.), Programming Languages and Systems: Proceedings of the 19th European Symposium on Programming (ESOP 2010), in: Lecture Notes in Computer Science, vol. 6012, Springer, 2010, pp. 125–144.

[30] D. Sannella, A. Tarlecki, Specifications in an arbitrary institution, Inf. Comput. 76 (2) (1988) 165–210, https://doi.org/10.1016/0890-5401(88)90008-9.

[31] M. Haveraaen, M. Roggenbach, Specifying with syntactic theory functors, J. Log. Algebraic Methods Program. 113 (2020) 100543, https://doi.org/10.1016/j.jlamp.2020.100543.

[32] J.A. Goguen, G. Rosu, Institution morphisms, Form. Asp. Comput. 13 (3–5) (2002) 274–307, https://doi.org/10.1007/s001650200013.

[33] J. Meseguer, General logics, in: H.-D. Ebbinghaus, J. Fernández-Prida, M. Garrido, D. Lascar, M. Rodríguez Artalejo (Eds.), Logic Colloquium'87, North-Holland, 1989, pp. 275–329.

[34] M.J. Sannella, Constraint satisfaction and debugging for interactive user interfaces, Ph.D. thesis, University of Washington, Seattle, WA, USA, 1994, UW Tech Report 94-09-10.

[35] G. Trombettoni, B. Neveu, Computational complexity of multi-way, dataflow constraint problems, in: Proceedings of the 15th International Joint Conference on Artifical Intelligence–vol. 1, IJCAI'97, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997, pp. 358–363.