Typelets — A Rule-Based Evaluation Model for Dynamic, Statically Typed User Interfaces

Martin Elsman[1] and Anders Schack-Nielsen[2]

[1] University of Copenhagen, Universitetsparken 5, DK-2100 Copenhagen, Denmark
mael@diku.dk
[2] SimCorp, Weidekampsgade 16, DK-2300 Copenhagen, Denmark
anders.schack-nielsen@simcorp.com

Abstract. We present the concept of *typelets*, a specification technique for dynamic graphical user interfaces (GUIs) based on types. The technique is implemented in a dialect of ML, called MLFi,[3] which supports dynamic types, for migrating type-level information into the object level, so-called type properties, allowing easy specification of, for instance, GUI control attributes, and type paths, which allows for type-safe access to type components at runtime. Through the use of Hindley-Milner style type-inference in MLFi, the features allow for type-level programming of user interfaces. The dynamic behavior of typelets are specified using declarative rules. The technique extends the flat spreadsheet programming model with higher-order rule composition techniques, extensive reuse, and type safety. A layout specification language allows layout programmers (e.g., end-users) to reorganize layouts in a type-safe way without being allowed to alter the rule machinery. The resulting framework is highly flexible and allows for creating highly maintainable modules. It is used with success in the context of SimCorp's high-end performance-critical financial asset-management system with screens containing several hundreds of GUI controls located in group-boxes, subtabs, and menu structures and with very complex dependency structures defined using declarative rule composition.

1 Introduction

Complex GUI applications are often developed using costly and error prone development procedures for which developers are required to design the precise static layout of GUI controls, using a so-called designer tool, and develop an excessive amount of boilerplate side-effecting event-handler functions for which the host language provides little (or no) type guarantees.

This paper presents a technique to obtain a dynamic GUI given a declarative description (in terms of a MLFi type declaration) that specifies the type of the different controls in the user interface as well as possible high-level layout properties such as relative positions and groupings of controls. The approach supports

[3] MLFi is a derivative of OCaml, extended by LexiFi with extensions targeted at the financial industry.

a large set of composable GUI controls, including ordinary value input fields (for integers, floats, amounts, etc.), buttons, select boxes, check boxes, date-picking controls, grid controls, and various grouping controls, such as labeled groups, tab controls, and more.

For specifying the dynamic behavior of a typelet, the programmer writes rules, stating, for example, that a change in some fields influence the content of other fields. The rule-based approach is declarative in the sense that focus is on "what the end-user gets" instead of "how the end-user gets it".

Rules may be composed and attached to a typelet in a type-safe way. Moreover, rules are objects for analysis in the sense that it may statically be determined, for instance, that different rules target the same field or that a subset of rules form a cyclic dependency. The declarative nature of typelet rules is similar to the Functional Reactive Programming (FRP) approach, as seen in Fran [7], Fruit [6], and Flapjax [16].

The typelet implementation is augmented with a type-safe and rule-preserving layout specification mechanism, that allows for layout programmers (e.g., the end-user) to freely reorganize layouts using a set of layout combinators.

The contributions of this paper are the following:

1. We present a novel technique for programming dynamic graphical user interfaces, based on the notion of types and declarative rules for specifying how different parts of the GUI interact.
2. We show how the technique can be augmented with a technique for separating layout from functionality.
3. We describe how this novel declarative approach to programming graphical user interfaces with success is used in practice in SimCorp Dimension, a financial asset-management system with typelet-based trade screens containing several hundreds of inter-dependent fields and other controls, such as grids.
4. Finally, the paper also serves to demonstrate the usefulness of some of the dynamic type features of MLFi, including type properties, and type paths.

We first present a simple typelet and proceed by showing how MLFi type properties may be used to control details of layout and GUI behavior. In Sec. 4, we outline the dynamic type features of MLFi. We then cover the central concept of rules in Sec. 5 and discuss some details of the implementation in Sec. 6. The augmented type-safe layout specification mechanism is described in detail in Sec. 7. Related work is presented in Sec. 8 and Sec. 9 concludes.

2 Typelet Basics

We first demonstrate the typelet idea with a simple example that allows a user to enter some personal data and information about whether the user has passed an introductory programming course. Fig. 1(a) lists a typelet specification for the user interface. The result of displaying the specification as a GUI is shown in Fig. 1(b).

```
type gender = Male | Female
type t = {
   name: string;
   address: string;
   age: int;
   gender: gender;
   passed_course: bool
}
```

(a)

(b)

Fig. 1. A simple typelet with data entered by the end user.

There are a series of points to be made here. First, notice that MLFi record field names are used as labels in the GUI; for localization purposes, the implementation allows these names to be overwritten by a resource file. Second, notice that default controls are selected based on the type of record field names in the typelet; for instance, a drop down selection box is chosen for the `gender` field. Third, the order the controls appear in the GUI matches closely the order of fields in the typelet. Finally, notice that default values are chosen for each control.

Typelets also support more dynamic behavior. For instance, if a typelet contains sum-types with data constructors that take arguments, a dynamic GUI is generated for which the GUI's representing the data constructor arguments are replaced and shown based on a left-positioned drop-down list with data constructor names.

There are many details to consider regarding the layout of even the simple typelet presented above. For instance, should all fields extend to the right if the GUI window is enlarged? How can it be specified that a radio button group is desired instead of a drop down selection box for the gender? How can it be specified that two controls should appear on the same row?

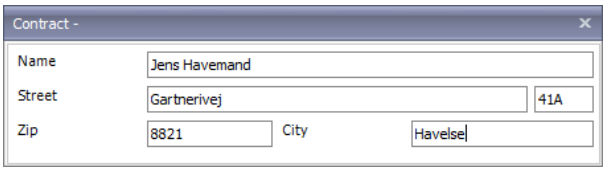
3 Increasing Control with Type Properties

MLFi supports the notion of type properties, which allows the programmer to attach arbitrary key-value properties (or key properties) to types. Fig. 2(a) lists the code for a small example typelet that makes use of type properties setting the width of controls and for specifying that a control should appear to the right of another control. The result of displaying the typelet is shown in Fig. 2(b).

A large number of type properties are supported for controlling the layout for various controls, including the number of digits for float fields, the caption field for a control, the height, width, and default value for a control, and so on. As demonstrated by the first two type declarations in the example, it is possible to make use of ML's type inference (and the fact that sets of type properties compose) to ease the annotation of types with type properties.

```
type 'a r = 'a + [right]
type 'a fixed = 'a + [fixedwidth]
type t = {
  name: string; street: string;
  no: string r fixed
    + [width="50";
       nocaption];
  zip: string fixed
    + [width="100"];
  city: string r
}
```

(a)



(b)

Fig. 2. Use of type properties to control layout.

Whereas this possibility is great for getting a good initial layout for a user interface, we shall see in Sec. 7 how so-called typelet layouts allow for separation of layout specification from functionality.

The typelet implementation makes use of special MLFi features (described in the next section) for computing a runtime representation for a type and for inspecting type properties in runtime representations of a type. Using these features, the typelet implementation allows for the typelet-programmer to make use of type properties for specifying details of how a control should be displayed and for specifying default values for controls, and so on.

4 Dynamic Types and Type Paths in MLFi

Before proceeding with presenting how a user may specify rules to give dynamic behavior to user interfaces, we summarize how MLFi extends OCaml with dynamic types and so-called type paths [11].

MLFi provides a universal datatype for representing static types at runtime:

```
type utype = Int | ...
           | List of utype | Option of utype
           | Record of (string * utype) list
           | Props of utype * (string * string) list
           | ...
```

Notice that the representation allows for the programmer to inspect the type properties for a type, inferred at compile time and provided to the programmer using the `Props` value constructor.

Further, MLFi supports an abstract notion of *typed dynamic types*, of type t `ttype`, for some concrete type t. Values of type t `ttype` can be constructed using the simple expression form `(ttype_of:`t`)` for injecting the static type t into a value of type t `ttype`. Values of type t `ttype` can easily be converted into values of type `utype`, with no computational overhead, using the function `to_utype: 'a ttype -> utype`. The type argument to the `ttype` type

constructor is really just a phantom type, which provides for improved type-safe programming [2, 9, 10, 15]. In concert with the support for dynamic types, MLFi supports the notion of a universally tagged representation of values, called *variants*, which are useful for programming ad-hoc polymorphic functions, with a "pay-as-you-go" strategy (no overhead forced on ordinary code). MLFi has a built-in function `variantize: t:'a ttype -> 'a -> variant` and another built-in function `devariantize: t:'a ttype -> variant -> 'a`, which may fail by raising an exception. Notice here the so-called *labeled arguments* `t:'a ty`, a special feature of MLFi, which allows for the programmer to label particular arguments. When calling such functions, labeled arguments can be provided explicitly, as in `variantize ~t:(ttype_of:int) 5` or implicitly, in the case of typed dynamic types, with the compiler looking in the context for a value of the particular inferred type. In many cases, the programmer can then omit the typed dynamic type arguments. In the case above, the programmer may simply write `variantize 5`. Using the above features, it is straightforward to write pseudo-ad-hoc polymorphic functions, such as `print: t:'a ttype -> 'a -> string`.

MLFi also supports the notion *type paths*, which are values representing functions for pointing at a subcomponent of a value. Type paths have type `(t,s)tpath` where `t` is a type containing `s` as a subcomponent. A special type path is the identity type path of type `(t,t)tpath` for arbitrary `t`.

Syntactically, type paths are written using dot-notation (with a prefix dot). As an example, if `{a:{b:int;c:string}; d:bool}` is a MLFi type `t`, then `.a.b` is a type path of type `(t,int)tpath`. Type paths are a little more than selector functions on types. They compose, using a type path compose operator, but it is also possible to extract from a type path, at runtime, the sequence of labels that define the type path. The runtime representation works well together with dynamic types and variant values.

5 Rules for Specifying Typelet Dynamics

Before we describe the concept of typelet rules in detail, we demonstrate the concept with a simple temperature typelet:

```
type 'a ro = 'a + [readonly]
type temp = {celsius: float; fahrenheit: float ro; kelvin: float ro}
open Fields
let calc =
  Rule.update (value(.celsius)) (value(.fahrenheit) & value(.kelvin))
    (fun c -> (9.0 /. 5.0 *. c +. 32., c +. 273.15)
let low =
  Rule.validate (value(.celsius))
    (fun c -> if c < -273.15 then Some "Temperature too low" else None)
let () = typelet "Temperature" ~t:(ttype_of:temp) ~rules:[low;calc] ()
```

Notice first the load of the typelet using the `typelet` function in the last line. This function takes as argument a name, the type of the Typelet (i.e., the argument for `~t`), and a list of rules. Notice also that the Fahrenheit and Kelvin fields

are marked `readonly` using type properties in the type declaration for t. The dynamic behavior of the typelet is specified using two rules, one that updates the Fahrenheit and Kelvin fields when there are changes to the Celsius field, and one that reports an error when a value in the Celsius field becomes invalid. The resulting typelet is shown in action in Fig. 3.

(a) (b)

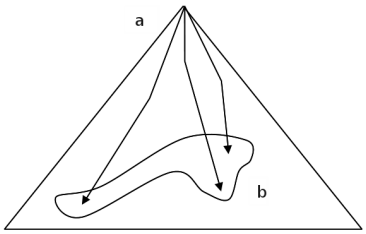
Fig. 3. Temperature typelet. Image (a) shows the typelet after evaluation of the update rule (upon change of the Celsius field). Image (b) shows the typelet after evaluation of the validate rule on invalid input (Celsius below -273.15 degrees).

In principle, the `Rule.update` function takes three arguments, (1) a specification of which source fields the rule depends on, (2) a specification of which target fields the rule targets, and (3) a MLFi function that accepts a value corresponding to the source specification and computes a result corresponding to the target specification. The source and target specifications are specified using an algebra over type paths. The algebra over type paths allows for selection of multiple fields and for referring to basic properties of a field, such as its value or whether the field is read only or disabled.

The module type for the `Fields` module is presented in Fig. 4(a).

```
module type FIELDS = sig
 type ('i,'a)t  (* 'i : type of the root *)
                (* 'a : type of elements pointed to *)
 val const    : t:'a ttype -> 'a -> ('i,'a)t
 val value    : ('i,'a)tpath -> ('i,'a)t
 val enabled  : ('i,_)tpath -> ('i,bool)t
 val readonly : ('i,_)tpath -> ('i,bool)t
 val restrict : ('i,'a)tpath
                -> ('i,'a list)t
 val (&)      : ('i,'a)t -> ('i,'b)t
                -> ('i, 'a*'b)t
end
```

(a)



(b)

Fig. 4. The `FIELDS` module type (a) and an example of a `fields` value composed of three fields (b).

A value of type `(a,b)fields` for some `a` and `b` represents a set of located fields inside the type `a`. The diagram in Fig. 4(b) illustrates a case where the `fields` value is composed of three fields within `a`.

The `const` function provides functionality for expressing a constant field value whereas the `value` function gives access to the content of a field. The functions `enabled` and `readonly` give access to a field's enabled property and read-only property, respectively (as boolean values). The `restrict` function makes it possible to refer to the restricted set of valid values for a field; when used in the target of a rule, the set of valid values for a field may be restricted dynamically.

The `&` operator may be used to compose field values, as we have seen in the example. Most of the functions in the `Fields` module takes a `ttype` argument. In normal use of the module, the arguments are passed implicitly by the compiler and the programmer need not be explicit about these arguments, as can be seen in the example above.

```
module type RULE = sig
 type 'i t
 type ('i,'a) fields = ('i,'a) Fields.t
 val update   : ta:'a ttype -> tb:'b ttype
                -> ('i,'a)fields -> ('i,'b)fields -> ('a -> 'b) -> 'i t
 val validate : t:'a ttype -> ('i,'a)fields
                -> ('a -> string option) -> 'i t
 val button   : ta:'a ttype -> tb:'b ttype
                -> ('i,'a)fields -> ('i,'b)fields
                -> ('a -> 'b) -> ('i,unit)tpath -> 'i t
 val grid     : ('i,'a)fields -> ('i,'b list)tpath -> ('a * 'b)t -> 'i t
 val grid_add : t:'a ttype -> ('i,'a)fields -> ('i,'b list)tpath
                -> ('b,'c)fields -> ('a -> 'c) -> 'i t
 val default  : t:'a ttype -> ('i,'a)fields -> (unit -> 'a) -> 'i t
 val subpath  : ('i,'a)tpath -> 'a t -> 'i t
 val all      : 'i t list -> 'i t
 val iso      : ta:'a ttype -> tb:'b ttype
                -> ('i,'a)fields -> ('i,'b)fields
                -> ('a -> 'b) -> ('b -> 'a) -> 'i t
 val weak_upd : ta:'a ttype -> tb:'b ttype
                -> ('i,'a)fields -> ('i,'b)fields -> ('a -> 'b) -> 'i t
 ...
end
```

Fig. 5. The `RULE` module type.

The module type for the `Rule` module is presented in Fig. 5. As we have seen earlier, the `update` function takes as arguments a source field specification, a target field specification, and an appropriate MLFi function that matches the source and target specifications. In addition, the function takes as argument two

`ttype` arguments. As described in Sec. 4, these arguments are provided implicitly whenever the call site context provides the appropriate values.

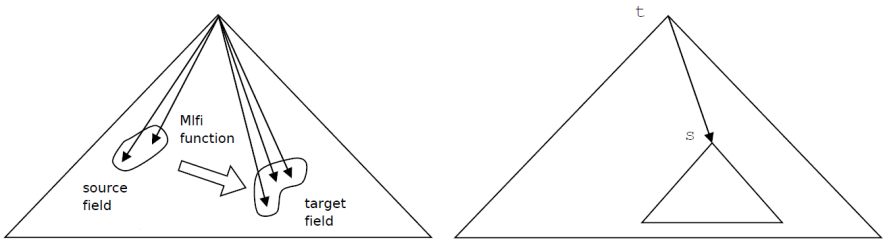


Fig. 6. Illustration of (a) the `update` rule and (b) the `subpath` rule.

Fig. 6(a) illustrates the semantics of the `update` rule. Intuitively, when a source field is modified, either by an end user or by another rule, the source values are extracted to form an argument for the MLFi rule function. Hereafter the function is applied and the result is stored into the target fields denoted by the target field specifier.

The implementation takes care that each rule is evaluated only once for each field modification performed by an end user. Rules are not allowed to form cycles, except through the `iso` and `weak_upd` rules (see below), thus rules may be topologically ordered. Instead of evaluating rules eagerly when triggered by a change in a source field, rules are dynamically added to a heap structure when a source field changes value. The heap structure is evaluated by repeatedly evaluating the topologically lowest ordered rule in the heap. Given that no cycles appears in the graph defined by the rules and given that each rule satisfies some validity constraints, the rule evaluation strategy guarantees that rules are evaluated on consistent data and that each rule is evaluated at most once in reaction to a field update.

The `validate` function takes only a source field specifier and a function that optionally returns an error message (besides from an appropriate `ttype` argument). The implementation guarantees that validate functions that have a specific field in its source field specifier are evaluated before other rules that have the same field in its source field specifier.

The `button` function takes four non-ttype arguments, (1) a source field specifier, (2) a target field specifier, (3) an evaluation function, to be evaluated when the button is pressed, and (4) a type path to a unit type, which serves to identify the button in the generated layout.

The `grid` function lifts a rule that works on a pair of auxiliary GUI data and data for a grid row to a rule that works on an entire grid, represented as a list of values (i.e., a list of rows). The `grid_add` function is used to specify field data for new rows added to the grid (by the user). The supplied type path points to the grid. The supplied function takes as argument data specified by the

first fields specifier. The result of the supplied function matches a field specifier relative to the data for a row in the grid. Those fields in the added row that are not mentioned in the relative field specifier are filled with default values.

The `default` function provides functionality for specifying default values other than the built-in defaults or defaults specified using type properties.

The `subpath` function makes it possible to lift a rule for some type `s` to a rule for a type `t` that contains `s` in the sense that there exists a type path from `t` to `s`. The relation between `t` and `s` is illustrated in Fig. 6(b). The `all` function makes it possible to treat a list of rules as one rule. These two functions are important for building new rules from existing ones.

The last two rule functions shown, `iso` and `weak_upd`, allow for certain kinds of cycles in the fields dependency graph formed by a particular set of rules. The `iso` rule allow the programmer to set up an isomorphism between fields—it is the obligation of the programmer to guarantee that the supplied functions form an isomorphism. The `weak_upd` rule function works like the `update` function, except that the rule is triggered only when the change in a source field is due directly to a modification by an end user. This latter function has proven to be useful, for instance, for implementing a generic "fill out utility" that allows a user to select a value in a dropdown box and thereby get the effect that a series of fields are filled out with computed data, but in such a way that if some value in the set of filled out fields is edited, the wittness in the dropdown-box is erased. By using weak update rules for both the "fill out" functionality and the erase functionality, cycles in the rule evaluation is avoided.

6 Implementation

The implementation of typelets in the SimCorp Dimension asset-management system, targets the .NET platform via an extension to Microsoft's Windows Forms library. Whereas all rules are specified and analyzed in MLFi, the Windows Forms control tree is generated at the .NET side based on a serialized variant-representation of the type that specifies the layout of the typelet. Besides from the control-tree, a container tree is also constructed at the .NET side based on the (variant-representation of the) typelet type. Once both the container-tree and the control-tree are constructed, the containers are bound to the controls, which has the effect that changes in the containers will have a visible effect in the controls. Information about rules is also serialized and communicated to the .NET side. For each update rule, for instance, event handlers are attached to the source controls, by traversing the GUI control structure using type-indexed functions that iterate on the variantized version of the relevant type paths. At runtime, an attached event handler will, when triggered, collect the argument represented by a fields value, serialize the argument into MLFi representation, call the registered MLFi function, and store the result in the fields represented by the target fields value.

The typelet mechanism is by no means tied to the .NET platform. If desired, it should be straightforward to replace the .NET part of the framework with,

for instance, a JavaScript/HTML backend using, for instance, SMLtoJs [8] or `js_of_ocaml` [23].

6.1 A Computation Monad

The MLFi runtime system is single-threaded and not reentrant, which make it impractical to let MLFi functions make queries to the database and call expensive functions (e.g., monte-carlo simulations for contract pricing [13]) on the .NET side. For this reason, the actual interface provided to the rule programmer is a slight modification of the `RULE` module type given in Fig. 5. The actual `RULE` module type exposes a monadic interface to computations [18], through a monad of type `'a m`. In effect, the type for the `update` rule function actually takes the following form:

```
val update : ta:'a ttype -> tb:'b ttype
             -> ('i,'a)fields -> ('i,'b)fields -> ('a -> 'b m) -> 'i t
```

Functionality on the .NET-side are exposed to the MLFi programmer as monadic computations, which may be composed with direct MLFi computations using the monad's `return` and `bind` functionality. Now, because the composed computations are driven from the .NET-side, the MLFi runtime system is blocking for entrance only when it is busy computing.

6.2 A Functional-Relational Mapping Scheme

The typelet implementation is also augmented with a typed functional-relational mapping scheme for mapping data in a typelet into a form acceptable for a relational database system and vice versa. The mapping forms an isomorphism between the data in the database and the data in the typelet and is used both for loading and saving typelet data. In this paper we have focused on the more dynamic behavior of typelets and we shall not discuss the functional-relational mapping scheme in more detail here, except by stating that the mapping scheme is applied for screens where the user may load particular stored data into the screen, either for presentation purposes or for the purpose of making changes to the data. Similarly, the mapping mechanism is used whenever data in a screen needs to be stored.

7 Separating Concerns Using Typelet Layouts

A front-end programmer may specify a complete redesign of a typelet using a set of combinators to form a so-called typelet layout. Besides from basic combinators for grouping controls in tab pages and group controls, two basic combinators are available, namely the `pick` combinator, which selects (using a type path) a component from the typelet (an entire group or a concrete control) and the `apply`

combinator, which replaces a subcomponent in a layout with an alternative layout. As we shall see, the typelet layout combinators are guaranteed not to alter the rule semantics of the underlying typelets.

The front-end programmer may choose to redesign the entire standard layout (as induced by the type for the underlying typelet) or use parts of the standard layout in the defined layouts. Typelet layouts are first class entities and there is no limit to the number of layouts that can be associated with a typelet. Typelet layouts are typed in the sense that they are defined for particular typelets (or typelet library components). The typing ensures that we can give appropriate meaning to a typelet layout.

Typelet programmers write typelet layouts in an embedded domain specific language for layouts. Fig. 7 lists the module type for the language.

```
module type LAYOUT = sig
  type 'i t                                  (* Layout for 'i-typelets *)
  type caption = string
  type halign = Left | Center | Right
  type valign = Top | Middle | Bottom

  val grp     : 'i t -> 'i t                 (* Grouping environment *)
  val (%%)    : 'i t -> 'i t -> 'i t          (* Horizontal sequencing *)
  val (@@@)   : 'i t -> 'i t -> 'i t          (* Vertical stacking *)
  val box     : caption -> 'i t -> 'i t      (* Wrap box around a layout *)
  val tab     : 'i t list -> 'i t            (* Show boxes as tabs *)
  val halign  : halign -> 'i t -> 'i t       (* Horizontal alignment *)
  val valign  : valign -> 'i t -> 'i t       (* Vertical alignment *)
  val hspace  : int -> 'i t                  (* Horizontal space *)
  val vspace  : int -> 'i t                  (* Vertical space *)
  val caption : caption -> 'i t -> 'i t      (* Use the provided caption *)
  val pick    : ('i,'a)tpath -> 'i t         (* Pick standard layout item *)
  val apply   : ('i,'a)tpath -> 'a t         (* Apply alternative layout *)
                -> 'i t -> 'i t

  (* Derived combinators *)
  val emp     : 'i t                         (* Empty layout *)
  val all     : 'i t                         (* Complete type layout *)
  val hide    : ('i,'a)tpath -> 'i t         (* Hide pointed-to item *)
                -> 'i t
  val lift    : ('i,'a)tpath -> 'a t         (* Lift pointed-to item *)
                -> 'i t
  val (%)     : 'i t -> 'i t -> 'i t         (* Padded sequencing *)
  val (@@)    : 'i t -> 'i t -> 'i t         (* Padded stacking *)
end
```

Fig. 7. The `LAYOUT` module type.

The grouping environment introduced by `grp` allows the layout programmer to organize layouts in a grid style with proper alignment of columns and rows. In a group environment (e.g., in an argument to `grp` or `box`), the programmer may use the `%` and `@@` combinators to separate items and rows (of items), respectively.

The alignment and space combinators give programmers control over the positioning of items without allowing programmers to work with absolute positioning. Notice that layouts should adapt properly to resizing of typelets and that layouts should position themselves properly, also on limited space.

The `pick` combinator allows the programmer to pick a layout from the type as pointed to by the type path argument. The `apply` combinator applies a given layout to a pointed-to item in a larger layout.

The `tab` combinator takes a list of boxes, which may either be constructed using the `box` combinator or picked from the typelet (by picking an existing tab element, a box, or an existing tab group).

It is possible, and often straightforward, to define derived combinators such as the `hide` and `lift` combinators. For instance, the `hide` combinator is implemented as follows:

```
let hide tp = apply tp emp
```

The interface imposes some restrictions. For instance, we have deliberately chosen not to allow the programmer to overwrite the default minimum size and width-flexibility of a control. Thus, picking a date control yields a date control with the same width, height, and caption as the picked control. Also, we do not attempt to capture, at the type level, which components are shown or whether an item is a box or another kind of object. This choice is deliberate; we want to keep the layout concept simple without cluttering the types with additional type parameters.

Fig. 8 demonstrates various features of the layout programming interface, including regrouping. Notice that the type `currency` is defined elsewhere as a sum datatype, which present themselves as a drop-down control.

It is natural to ask for properties of the `pick` and `apply` combinators. In particular, we would expect the following property to hold:

Property 1. For all type paths p, it holds that `all` = `apply` p (`pick` p) `all`.

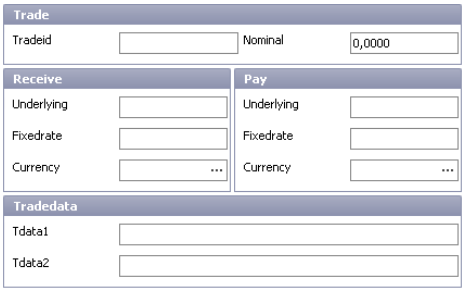
Typelet layouts may be registered with the typelet at typelet definition time or loaded and linked dynamically using MLFi's dynamic linking features. Fig. 9 shows a layout for an input screen for an interest rate swap, a complex financial instrument used by most financial institutions for hedging interest rate risk. Data can be entered by the user in any order and the rule machinery calculates a number of derived values whenever sufficient information is typed in by the user.

8 Related Work

There is a large body of related work. One strand of related work includes work on providing type-safe language bindings for constructing graphical user

```
type leg = {legno: int; underlying: string option;
            fixedrate: float option; currency: currency}
type tradedata = {tdata1: string; tdata2: string}
type tlet = {tradeid: string; nominal: float;
             receiveleg: leg; payleg: leg; tradedata: tradedata}
open Layout
let leg = pick(.underlying) @@
          pick(.fixedrate) @@
          pick(.currency)
let l2 : tlet t =
  pick(.tradeid) % pick(.nominal) @@
  grp(box "Receive"
        (lift(.receiveleg)leg) %
      box "Pay"
        (lift(.payleg)leg)) @@
  pick (.tradedata)
```

(a)



(b)

Fig. 8. An example typelet layout (a) and it's effect on the typelet presentation (b).

interfaces in functional languages [3, 19, 14] either using monads or by using the effectful features of a language for controlling the behavior of a GUI. A specific monadic combinator library for constructing GUI's is the Clean iTask library [17, 20], which primarily focuses on allowing the programmer to generate a workflow GUI from a declarative specification of the GUI and the workflow. Compared to the iTask framework, typelets do not address how windows are opened and closed, but rather on how fields, grids, and controls change upon changes in a field.

Like the typelet library, many GUI libraries make use of phantom types [2, 9, 15] as a mechanism for providing increased type-safety, for instance through modeling single-inheritance [10]. Phantom types are used in the typelet implementation both for the `Fields`, `Rule`, and `Layout` modules to restrict the composability of values.

Another branch of related work is the large body of work on functional reactive programming [5–7, 16, 22], which has served as inspiration for the rule mechanism for typelets. In particular, using a topological ordering of rules and a heap data structure to guarantee that rules are triggered at most once upon a change of input is directly influenced by previous work on implementations of functional reactive programming [8]. The work on flowlets [1] combines work on functional reactive programming with formlets [4], which, as typelets, focuses much on composability of GUI components.

Other related work investigates the possibility of synthesizing user interfaces and event handling code for interdependent fields based on formal descriptions specified by the programmer in a domain specific language for specifying the logic dependencies. Both the work on property models [12] and the work on Plato [21], a compiler for interactive web forms, follows this direction. In the typelet

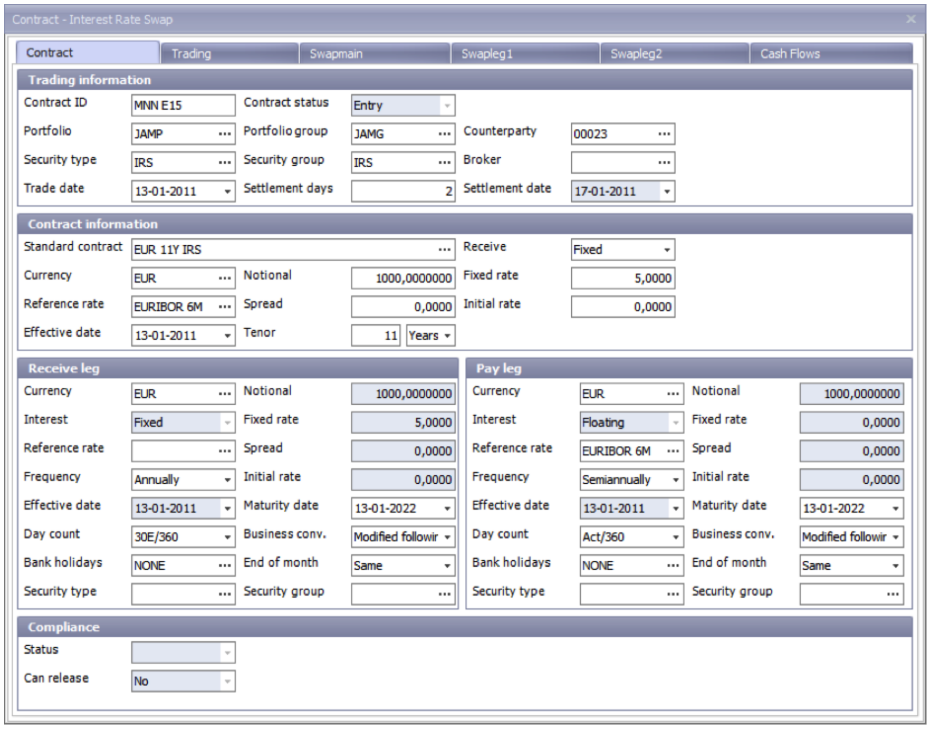


Fig. 9. Input screen for an interest rate swap, a complex financial instrument used by most financial institutions for hedging interest rate risk.

approach, cyclic dependencies are only supported in a controlled way, through iso-rules and weak rules, and programmers need to be explicit about such cyclic dependencies, which makes it straightforward to express to programmers the requirements for composing user interface components.

9 Conclusion

We have presented the concept of typelets, which have been designed for constructing trade screens for the SimCorp Dimension asset management system. Each trade screen can have more than 400 individual fields located in nested tab-structures and group controls. Together with a functional-relational mapping (for storing and loading database content), the typelet implementation forms a dynamic GUI mechanism, which is declarative and statically typed, but also highly flexible.

Acknowledgments

The authors want to thank the Instrument Modelling Language team at SimCorp for many interesting discussions and the PADL'14 reviewers for their helpful and insightful feedback. This research has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the research center HIPERFIT: Functional High Performance Computing for Financial Information Technology (`hiperfit.dk`) under contract number 10-092299.

References

1. Joel Bjornson, Anton Tayanovskyy, and Adam Granicz. Composing reactive GUIs in F# using WebSharper. In *Proceedings of the 22nd International Symposium on the Implementation and Application of Functional Languages (IFL'10)*, pages 203–216. Springer-Verlag, 2011.
2. Matthias Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively.". In *Workshop on Multi-language Infrastructure and Interoperability (BABEL'01)*, September 2001.
3. M. Carlsson and T. Hallgren. Fudgets–a graphical user interface in a lazy functional language. In *Proceedings of the ACM Conference on Functional Programming and Computer Architectures (FPCA'93)*, pages 321–330. ACM Press, 1993.
4. Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. The essence of form abstraction. In *Sixth Asian Symposium on Programming Languages and Systems (APLAS'08)*, 2008.
5. A. Courtney, H. Nilsson, and J. Peterson. The Yampa arcade. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 7–18. ACM Press, 2002.
6. Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Proceedings of the 2001 Haskell Workshop*, September 2001.
7. Conal Elliott and Paul Hudak. Functional reactive animation. In *ICFP'97*, New York, NY, USA, 1997. ACM.
8. Martin Elsman. SMLtoJs: Hosting a Standard ML compiler in a web browser. In *Proceeding of ACM SIGPLAN 2011 International Workshop on Programming Language And Systems Technologies for Internet Clients (PLASTIC'2011)*. ACM Press, October 2011.
9. Martin Elsman and Ken Friis Larsen. Typing XHTML Web applications in ML. In *International Symposium on Practical Aspects of Declarative Languages (PADL'04)*. Springer-Verlag, June 2004.
10. Matthew Fluet and Riccardo Pucella. Phantom types and subtyping. In *International Conference on Theoretical Computer Science (TCS'2002)*, August 2002.
11. Alain Frisch. Runtime types. In LexiFi blog, December 2011. Slides available from `http://www.lexifi.com/blog/runtime-types`.
12. Jaakko Järvi, Mat Marcus, Sean Parent, John Freeman, and Jacob N. Smith. Property models: from incidental algorithms to reusable components. In *7th International Conference on Generative Programming and Component Engineering (GPCE'08)*, pages 89–98, October 2008.
13. Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering. In *Fifth International Conference on Functional Programming (ICFP'00)*, September 2000.

14. Daan Leijen. wxHaskell: A portable and concise GUI library for Haskell. In *Proceeding of the 2004 ACM SIGPLAN Haskell Workshop*. ACM Press, September 2004.
15. Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *ACM Conference on Domain-specific languages*. ACM Press, 2000.
16. Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for ajax applications. In *OOPSLA'09*, New York, NY, USA, 2009. ACM.
17. Steffen Michels, Rinus Plasmeijer, and Peter Achten. iTask as a new paradigm for building GUI applications. In *Proceedings of the 22nd International Symposium on the Implementation and Application of Functional Languages (IFL'10)*, pages 153–168. Springer-Verlag, 2011. Selected papers.
18. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1989.
19. R. Noble and C. Runciman. Gadgets: Lazy functional components for graphical user interfaces. In *Seventh International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'95)*, pages 321–340. Springer-Verlag, September 1995.
20. Rinus Plasmeijer, Peter Achten, Pieter Koopman, Bas Lijnse, Thomas van Noort, and John van Groningen. iTasks for a change—type-safe run-time change in dynamically evolving workflows. In *Proceedings of the 20th International Workshop on Partial Evaluation and Program Manipulation (PEPM'11)*, pages 151–160. ACM Press, 2011.
21. Ricardo Rocha and John Launchbury. Plato: A compiler for interactive web forms. In *International Symposium on Practical Aspects of Declarative Languages (PADL'11)*. Springer-Verlag, 2011.
22. Meurig Sage. FranTk a declarative GUI language for Haskell. In *Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming (ICFP00)*, pages 106–117. ACM Press, 2000.
23. Jérôme Vouillon. Js_of_ocaml. Documentation at `http://ocsigen.org/js_of_ocaml/manual/`.