Static and Dynamic Visualisations of Monadic Programs

Jurriën Stutterheim
Institute for Computing and Information Sciences
Radboud University Nijmegen
P.O. Box 9010, 6500 GL
Nijmegen, The Netherlands
j.stutterheim@cs.ru.nl

Peter Achten
Institute for Computing and Information Sciences
Radboud University Nijmegen
P.O. Box 9010, 6500 GL
Nijmegen, The Netherlands
p.achten@cs.ru.nl

Rinus Plasmeijer
Institute for Computing and Information Sciences
Radboud University Nijmegen
P.O. Box 9010, 6500 GL
Nijmegen, The Netherlands
rinus@cs.ru.nl

ABSTRACT

iTasks is a shallowly embedded monadic domain-specific language written in the lazy, functional programming language Clean. It implements the Task-Oriented Programming (TOP) paradigm. In TOP one describes, on a high level of abstraction, the tasks distributed collaborative systems and end users have to do. It results in a web application that is able to coordinate the work thus described. Even though iTasks is defined in the common notion of "tasks", for stake holders without programming experience, textual source code remains too difficult to understand. In previous work, we introduced Tonic (Task-Oriented Notation Inferred from Code) to graphically represent iTasks programs using *blueprints*. Blueprints are designed to bridge the gap between domain-expert and programmer. In this paper, we add the capability to graphically trace the dynamic behaviour of an iTasks program at run-time. This enables domain experts, managers, end users and programmers to follow and inspect the work as it is being executed. Using *dynamic blueprints* we can show, in real-time, who is working on what, which tasks are finished, which tasks are active, and what their parameters and results are. Under certain conditions we can predict which future tasks are reachable and which not. In a way, we have created a graphical tracing and debugging system for the TOP domain and have created the foundation for a tracing and debugging system for monads in general. Tracing and debugging is known to be hard to realize for lazy functional languages. In monadic contexts, however, the order of evaluation is well-defined, reducing the challenges Tonic needs to overcome.

CCS Concepts

•**Software and its engineering** → **Functional languages;** *Source code generation;* **Runtime environments; Visual languages;** *System description languages;*

IFL '15, September 14-16, 2015, Koblenz, Germany

DOI: http://dx.doi.org/10.1145/2897336.2897337

Keywords

Dynamic program visualisation; purely functional programming; monads; iTasks; Clean

1. INTRODUCTION

When developing non-trivial software, one frequently needs to gather the correct requirements and frequently evaluate whether the right software is being built. This can be a hard and time-consuming activity when stakeholders with different backgrounds are involved. This is in part due to the communication gap that exists between experts in unrelated fields.

Task-oriented programming (TOP) is a style of functional programming that, amongst other things, aims to reduce the communication gap between various parties by developing programs in terms of the common notion of *tasks*. TOP is implemented by iTasks [11], a *shallowly embedded* monadic domain-specific language in the general-purpose, lazy, purely functional programming language Clean [12]. iTasks' combinators are used to compose multi-user web-based applications. Common technical issues related to distributed client-server settings, such as communication, synchronization, user interface generation, and interaction handling, are handled automatically by applying advanced functional programming techniques. These include type driven generic functions, and the ability to store, load and communicate closures in a type safe way using Clean's dynamic system.

As a result, an iTasks application writer is able to concentrate on the main issues: the tasks that have to be done by the end users in collaboration with their computer systems. Although an iTasks application writer can now concentrate on the things that matter, there still exists a communication gap between various stakeholders. Commonly, domain experts, managers and end users are not accustomed to reading and understanding textual source code. They prefer pictures, diagrams and natural language instead. Yet it is vital that they are able to evaluate the software that has been built, preferably more quickly than by simply running the program in a testing or production environment.

One way to bridge the communication gap between stakeholders and programmers is to utilise graphical notations. Well-known examples of such notations are BPMN [7] and UML [10]. However, such notations have as disadvantage that they are not part of the actual implementation and cannot practically be used as such. Additionally, since they are not part of the implementation, commonly manual labour is required to keep the models synchronized with the implementation. In practice, these models are prone to be-

coming outdated, because the cost of maintaining them may be higher than the benefit gained from the up-to-date documentation [2].

In previous work [18] we introduced our own graphical notation, called Tonic (Task-Oriented Notation Inferred from Code). Rather than specifying programs graphically, however, we made a specialised version of the Clean compiler, the Clean-Tonic compiler, which *generates* a graphical representation, called a *blueprint*, of the tasks that have been defined in Clean. Since blueprints are generated, they always provide up-to-date documentation of the source code. Implementing Tonic in the Clean compiler is necessary, since iTasks is shallowly embedded in Clean. As a result, programmers can use any Clean language construct to write iTasks programs. Implementing Tonic in the Clean compiler allows us to capture these language constructs in the blueprints we generate.

It is neither practical nor informative to show all the details of the original source code in the blueprints; they would become huge and unreadable. Instead we abstract from certain details yet provide enough information such that one should be able to understand by looking at the pictures which tasks have been defined and understand how these tasks depend on each other. We hope that by doing so, blueprints are easier to understand for non-programmers than the (Clean) code they are generated from.

The first incarnation of the Clean-Tonic compiler, however, did suffer from a number of drawbacks. For one, we could only generate *static* blueprints. Secondly, it had a hard-coded connection between the compiler and iTasks, which is not desirable for a general-purpose compiler. Thirdly, since the compiler was modified specifically for iTasks, Tonic's features were not usable in other contexts. Lastly, there was no way to customize the rendering of specific tasks without modifying both the compiler and iTasks.

In this paper, we set out to solve all of the aforementioned problems. We transform monadic programs such that dynamic information can be added to blueprints at run-time, creating *dynamic blueprints*. With these one can monitor what is happening with the monad during execution. In principle this can be done for any monad, but some programming effort is required to link its execution at run-time to the blueprints generated at compile-time. Our focus in this paper is one specific yet challenging use-case: the dynamic blueprints for iTasks, which is a highly complex and dynamic system.

iTasks is a challenging example because it is used for developing complex distributed systems. In the real world, people and systems often don't do their work as planned. Therefore it would be of great help if one were able to inspect what is going on at run-time. This aids, for example, programmers in debugging, domain experts in seeing whether the application works as designed, and managers and end users in tracking progress of workflows.

In essence, we have developed a kind of monitoring, tracing and debugging tool. This is commonly known to be a very challenging tool to make for a lazy functional language, particularly if one realizes that Clean applications are not interpreted, but compiled.

The Clean compiler is a state-of-the-art compiler, well known for the efficient code it generates. Due to the many transformations performed by the compiler to obtain such efficient code, and the laziness of the language, it is in general near to impossible to relate the execution of an application to a specific part of the original source code. The advantage we have here is that, since we restrict ourselves to monadic contexts, we statically know their order of evaluation.

A particular challenge is relating run-time behaviour to the corresponding parts of static blueprints. The difficulty comes from run-time calculations and higher-order functions. To do so, we modify the generated code by adding wrapper functions to monadic function applications. These wrappers tell which part of the original source code is being evaluated, so that it can be related to the correct part of the static blueprint.

With the dynamic blueprints we can show, at run-time, for any iTasks application, dynamic aspects such as: which tasks have been started, which are finished, which are running, how are they instantiated, what are the actual arguments, who is working on what, and which information is currently being produced by a specific task. The graphical representation of dynamic blueprints has to be modified at run-time to reflect the current program state.

In this paper we address the issues mentioned above and make the following contributions:

- We generalise the notion of blueprints to not only capture iTasks programs, but monadic programs in general. Using this new-found generalisation, we remove the hard connection between the Clean-Tonic compiler and iTasks, making Tonic a general solution.
- We show how static and dynamic blueprints are being made for the Task monad. Furthermore we discuss how our approach can be used for any monad, such as e.g. the IO Monad.
- We explain what kind of code transformations are made by the compiler such that we are able to map run-time behaviour to static information generated from the source program.
- We explain how we created a Tonic Task which allows an end user of any iTasks application to browse through the dynamic blueprints, and to inspect values of arguments and results of any task executed in the past or currently under evaluation.
- We explain that with a simple control-flow analysis and code transformation we can show the reachability of information (monads/task) in the blueprints. In this way we are able to show which future task can or cannot be executed given the current state of affairs.
- Tonic's end users can now customize the rendering of tasks using the declarative `Graphics.Scalable` library [1].

The rest of this paper is structured as follows: Section 2 shows several examples of static blueprints, after which Section 3 shows how these are made. Section 4 shows how we instantiate blueprints at run-time and incorporate run-time information in them, using an example in iTasks. Finally, Section 5 discusses related work, and Section 6 discusses current challenges and concludes.

2. EXAMPLES OF STATIC BLUEPRINTS

In this section we explain, using a number of examples, what kind of *static blueprints* we generate from Clean source

text. All blueprints in this paper are the actually generated blueprints for the example programs. Static blueprints have evolved notably since the previous Tonic paper. They consist of far fewer primitive shapes and are rendered completely declaratively using our `Graphics.Scalable` library.

In the introduction we already made clear that it is not a good idea to turn a complete Clean program into a graphical counterpart. First of all, Clean, much like Haskell, contains many language constructs. A pictorial representation isomorphic with the source code would be huge and not contribute to a better understanding of the code than the source program text itself. Secondly, there are technical obstacles that currently prevent us from showing all language features in a meaningful, graphical way. This is due to the fact that the Clean compiler generates highly efficient code, applying many transformations in the process. Some of the original code is simply no longer available.

Therefore, we decided to restrict ourselves to generating graphical representations for certain top level abstractions and a limited number of language primitives. We want to capture the major structure of the application being defined; we don't want nor need to provide all details of the application. We therefore decided to focus on monads. Monads are a frequently used abstraction in functional programming. In Haskell, for example, the `IO` monad is the principal way to perform side-effecting operations. As well as being useful, monads provide the ability to hide tedious book-keeping operations under the bind combinator, making code easier to read and reason about. In addition, the evaluation order of sequentially composed monadic computations is well defined and strict. The laziness of the language does not provide problems here.

We distinguish two sets of monads: one for which we want to generate a blueprint, and one for which we don't, but that may be *part* of a blueprint. We call the first set *blueprint* monads and the second *contained* monads. A blueprint monad is always a contained monad, but not the other way around. For iTasks, for example, the set of blueprint monads contains the `Task` monad, while the set of contained monads contains both the `Task` and the `Maybe` monad.

What makes generating blueprints challenging is that in any combinator definition, any Clean language construct may be used as well. As explained above, and further illustrated in the examples below, we limit ourselves to Clean language constructs which we are able to visualize in a meaningful manner, and hide those which are too complicated to visualize. We support **if**-blocks, **case**-blocks, pattern matching, **let**-blocks, recursion, higher-order functions and list definitions in the case that the number of list elements are statically known. For all other language constructs and cases we do not offer special graphical support in a blueprint. If we cannot graphically represent an expression, we pretty-print the original source code. Let's look at some examples.

2.1 Static Blueprints of the I/O Monad

The example below shows a simple interactive program implemented in Clean's IO monad (Clean does not have `do`-notation, so binds are explicitly written out). It asks the user to enter a number, confirms which number has been entered and then tells the user whether the number is prime or not. An example of its output is shown in Figure 1.

```
primeCheck :: IO ()
primeCheck =   putStrLn "Enter␣number:"
  >>|          getLine
  >>= \numStr. putStrLn ("Entered:␣" +++ numStr)
  >>|          if (isPrime (toInt numStr))
                 (putStrLn ("Is␣prime:␣" +++ numStr))
                 (putStrLn ("Isn't␣prime:␣" +++ numStr))
```

```
Enter number:
42
You have entered 42
42 is not prime
```

Figure 1: Example of IO performed by `primeCheck`

Figure 2(**a**) shows the blueprint we generate for this program. The graphical representation of the top-level `primeCheck` computation acts as a container for the other graphical elements. Each `IO` function application is represented by its own function-application box. The applied function's name is presented in bold on the top of the box, while its arguments are presented below it. Binds are represented by edges between two boxes. If the right-hand side of a bind is a lambda, the expression in the lambda is pretty-printed as edge-label.

2.2 Static Blueprints of the Task Monad

In this subsection we look at several example iTasks programs and the blueprint we generate for them. This subsection's goal is to give an intuition for Tonic and its blueprints, while at the same time explaining the basics of iTasks.

2.2.1 Prime Number Checker

An iTasks version of the `primeCheck` program is shown below, with its output shown in Figure 3 and its corresponding blueprint shown in Figure 2(**b**). Since iTasks is *shallowly* embedded in Clean, *all* Clean language features can be used to construct iTasks programs. Some of these, e.g. conditionals, we also want to include in the blueprints. Tasks are defined as functions with monadic result type `(Task a)` for some `a`. Sequential task composition is accomplished with the monadic bind combinator. `enterInformation` and `viewInformation` are examples of basic predefined *editor* tasks, which generate a web-based graphical user interface for a given type using generic programming techniques. The former editor allows the user to enter data using generically generated web forms, while the latter editor renders a read-only representation of the data.

```
primeCheck :: Task String
primeCheck =  enterNumber
  >>= \num. let numStr = toString num in
            viewInformation "Entered:" numStr
  >>|       if (isPrime num)
              (viewInformation "Is␣prime:" numStr)
              (viewInformation "Isn't␣prime:" numStr)
  where
  enterNumber :: Task Int
  enterNumber = enterInformation "Enter␣number"
```

Despite the fact that the previous two programs are defined in different monads, their blueprints are similar, since they share the common abstraction level of a monad.

2.2.2 Recursion and Higher-Order Tasks

iTasks' bind combinator automatically adds a "Continue" button to the user interface to progress to the right-hand

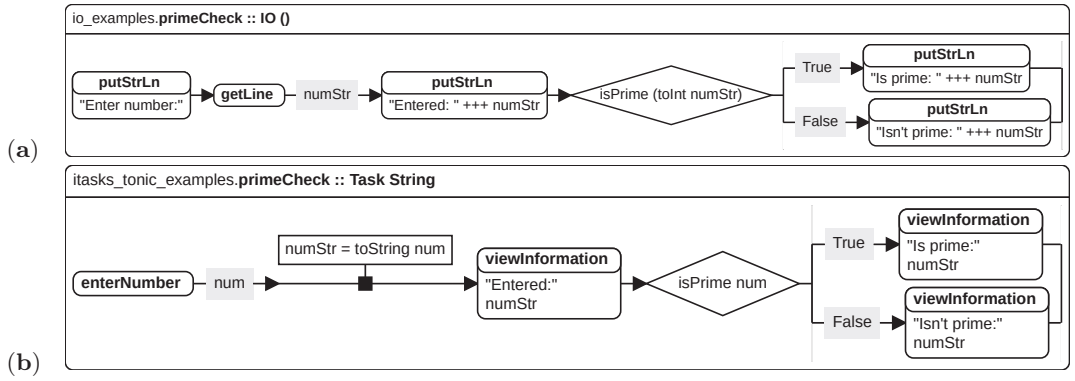

Figure 2: Static blueprint generated from the IO Monad example and its iTasks counterpart

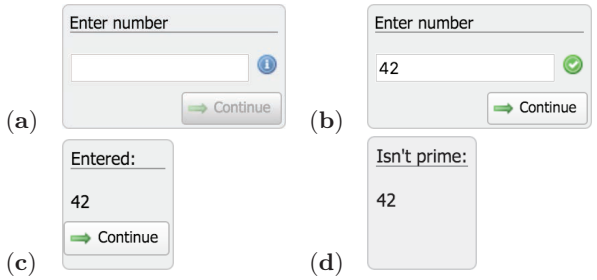

Figure 3: Example of the web forms generated by `primeCheck`

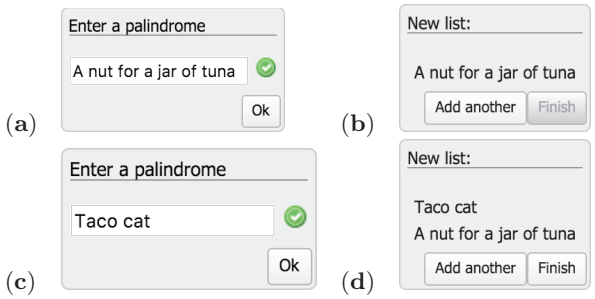

Figure 4: GUIs when applying `add1by1` to the `palindrome` task

side task. User-definable buttons can be created by using the *step* combinator (`>>*`), shown below (with its output in Figure 4(**a**). The step's left-hand side is a task that is executed first, while its right-hand side is a *list* of conditions paired with a follow-up task. If a condition is met, the corresponding follow-up task is executed.

```
add1by1 :: (Task a) [a] -> Task [a] | iTask a
add1by1 task listSoFar
  =          task
  >>= \elem. let newList = [elem : listSoFar] in
             viewInformation "New␣list:␣" newList
  >>*        [ OnAction (Action "Add␣another" [])
                        (always (add1by1 task newList))
             , OnAction (Action "Done" [])
                        (ifValue hasManyElems return))
             ]
  where
  hasManyElems :: [a] -> Bool
  hasManyElems xs = length xs > 1

addPalindromes :: Task [String]
addPalindromes = add1by1 palindrome []

palindrome :: Task String
palindrome
  =   enterInformation "Enter␣a␣palindrome"
  >>* [OnAction ActionOk (ifValue isPalindrome return))]
  where
  isPalindrome :: String -> Bool
  isPalindrome s = let s‘ = [toLower c \\ c <-: s | c <> ’␣’]
                   in  s‘ == reverse s‘
```

In this example two such conditions are provided. Both are an example of the `OnAction` condition, which causes a button to be rendered in the left-hand side task's user interface. `OnAction` takes two arguments: an *action*, which describes the button's text and a list of button meta-data, and a continuation to proceed to the next task once the corresponding button is pressed. The continuation is of type `(TaskValue a) -> Maybe (Task b)`. If the continuation returns `Nothing` the button is disabled, if it returns `Just`, the button is enabled and pressing it will progress the work-flow to the inner `Task b`.

Several convenience functions are available to write these continuation functions. In this example, `always` and `ifValue` are used. The former causes the button to be always enabled, while the second takes a predicate (`hasManyElems`) over the left hand-side task's value, enabling the corresponding button only if the predicate returns `True`.

Step functions are rendered differently from binds, as is shown in `add1by1`'s blueprint in Figure 5. Each condition in the step's right-hand side's list is rendered in its own branch. Continuation convenience functions as found in iTasks' standard libraries are rendered in a special way as well. Here, the `hasManyElems` predicate is rendered as a diamond, implying that this condition should be met before the work-flow can continue. The action is rendered as well, together with a small figure showing that it relates to a user action. It is possible to customize the way blueprints are rendered (see Section 4.2).

Common functional programming concepts, such as higher-order functions, recursion, and parametric polymorphism,

can be used as well. The `add1by1` task also demonstrates the use of common functional language features in the context of iTasks. The task has two arguments; a higher-order task `task` of type `(Task a)`, and an accumulator `listSoFar` of type `[a]`. On demand of the end user, `add1by1` recursively evaluates the higher-order task and accumulates the results. When finished, the accumulator is yielded as result. Notice that `add1by1` is not polymorphic in `a`, but overloaded. In Clean, context restrictions are specified at the end of a type definition (`| iTask a`). This context restriction is synonymous for several generic functions that take care of the type driven rendering of GUIs and the communication between server and client. This can automatically be derived by the Clean compiler for any first-order type. Context information is considered to be too much detail to mention in a blueprint and is therefore left out in the types shown there.

The higher-order task `task` is executed first. Statically we only know the type of `task`, but we do not know what its concrete value will be. For this we use a dashed frame. We do know that the task yields a value of proper type `a`. This value can be added to the accumulator (when the "Add another" button is pressed), after which `add1by1` recursively calls itself. Alternatively, the task can be terminated by pressing "Done", but this option can only be chosen when at least two values are collected in the list.

2.2.3 Parallel Tasks

Since lists are the most frequently used data structure in a functional language, several convenient language constructs are offered in Clean to handle them, such as dot-dot notation and list comprehensions. In general, one cannot statically deduce how many elements are contained in a list. In a static blueprint we therefore only show the elements of a list when it is statically known how many elements it contains. This holds for the list of step continuations used in the `add1by1` task, but it does not hold for the list of chat tasks used in the `parallelChat` task. `parallelChat`'s code is shown below and its blueprint in Figure 6.

```
parallelChat :: Task [[String]]
parallelChat
  =             get currentUser
  >>= \me.      enterSharedMultipleChoice
                  "Select␣friends" users
  >>= \friends. let users = [me : friends] in
                withShared (repeatn (length users) "")
    (\chatBox. allTasks (chatTasks users chatBox))
  where
  chatTasks :: [User] (Shared [String]) -> [Task [String]]
  chatTasks users chatBox = [  talk user i users chatBox
                            \\ i <- [0 .. ] & user <- users ]
  talk :: User Int [User] (Shared [String]) -> Task [String]
  talk user i users chatBox = user @: makeChat i users chatBox

makeChat :: Int [User] (Shared [String]) -> Task [String]
makeChat i users chatBox
  =   updateSharedInformationWith [selectChat i]
        (users !! i +++>  "is␣chatting:␣") chatBox
  ||- viewSharedInformationWith [dropChat i] "with:␣" chatBox
  where
  selectChat i
    = UpdateWith (\chatBox. chatBox !! i)
        (\chatBox chat. updateAt i chat chatBox)
  dropChat i
    = ViewWith (\chatBox.
        [  user +++> "␣says␣:␣" +++> chat
        \\ (user, chat) <- removeAt i (zip2 users chatBox)])
```

In `parallelChat`, the user of the task (`currentUser`) starts a chat by first selecting n friends to chat with from a list of administrated users. Next, $n+1$ `makeChat` tasks are started in parallel using the library combinator `allTasks`. This function expects a lists of tasks to be executed in parallel and ends when all its tasks are ended. In this particular example it is statically undecidable how many parallel task there will be, since it depends on the number of chosen friends. We will later see that at run-time we can in fact show these tasks in a dynamic blueprint, and see who is chatting with whom and inspect what they are chatting about.

Each `makeChat` task enables user i to have a chat with the others via a shared data source `chatBox` of type `Shared [String]`. Although Clean is a pure functional language, Shared Data Structures (SDS) can be modelled using iTasks. The shared list used here contains as many strings as there are chatting users, where the i-th element of the list represents the information typed in by the i-th chat user. In iTasks, shared data structures are maintained automatically. Whenever someone is changing the content of a shared data structure, any task that is looking at its structure is informed and updated automatically. This notification system works for any first-order data type, not just shared strings of text. In this example, chatting users automatically see what is written by someone else. Chat users can only update their part of the shared structure. In `updateSharedInformation` the i-th element is selected (`selectChat`) to be updated in the functions defined in `UpdateWith` while in `viewSharedInformation` the other elements are selected (`dropChat`) in `ViewWith` and shown read-only.

2.3 Blueprint Scalability

The examples in this section are relatively small. This is mostly due to the fact that most task definitions are small in practice. A lot of functionality can be obtained with very little iTasks code. While this shows that iTasks programs are well suited for visualization, it leaves the question whether blueprints scale well. In Tonic's current implementation, blueprints grow to the right when a monadic bind is used, while blueprints grow vertically when a **case** expression, parallel combinator, or step combinator is used. Blueprints are easy to layout, because all programs are trees, rather than general graphs. Other than the browser's scrolling mechanism, no additional navigational aids are available to view a blueprint, however. In theory this means that a blueprint for a big function can become unwieldy. In practice, however, blueprints for iTasks programs tend to stay small, even for non-trivial programs. As a result, we have not tried to solve this problem yet. Doing so is future work.

3. BUILDING STATIC BLUEPRINTS

Figure 7 shows the architecture of the modified Clean-Tonic compiler. In addition to the code the compiler normally generates (Intel, Arm and JavaScript), it now also generates a file containing blueprint information for each Tonic-enabled Clean module. This information can be read in by a special tool, the *Tonic Viewer*. The viewer is implemented in iTasks itself and can render blueprints in any HTML5 compatible browser.

As explained in the previous section, not all functions are automatically turned into a blueprint, only those with a blueprint monadic return type. Likewise, not all monadic function applications are turned into blueprint nodes, only those with a contained monadic return type. To differen-

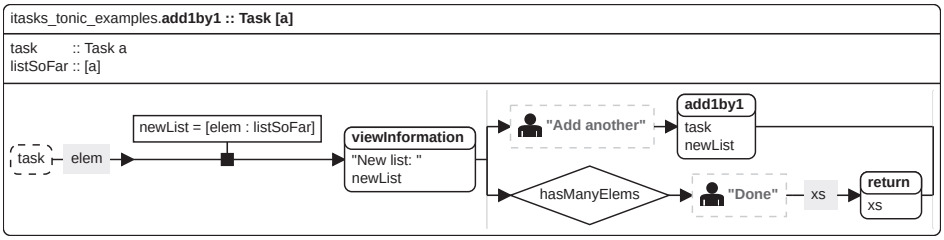

Figure 5: Static blueprint of `add1by1` with a higher-order task and recursion

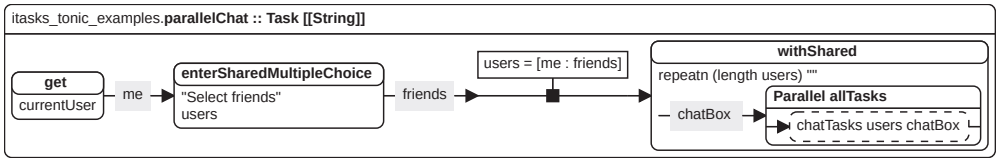

Figure 6: Static blueprint of parallel chat example in iTasks

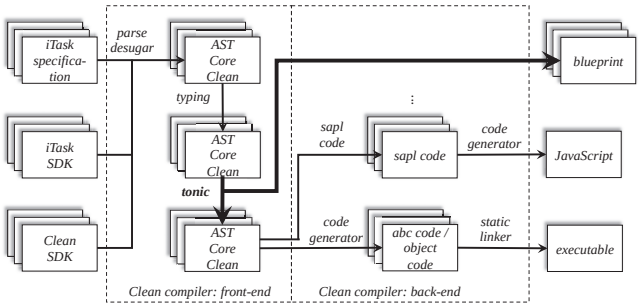

Figure 7: Global Architecture of the Clean - Tonic compiler

tiate between these sets of monads we introduce two type classes, `Blueprint` and `Contained`, shown below.

```
class Contained m | Monad m

class Blueprint m | Contained m
```

Whenever a programmer provides an instance of the `Blueprint` class for a certain type, a blueprint is generated by the compiler for every function which returns a monad of that type. Whenever an instance of the class `Contained` is provided, the application of the function in a blueprint is treated special. Any type with a `Blueprint` instance also requires a `Contained` instance, which is enforced by the former class' context restriction. Not all modules are considered for blueprint generation. Only modules that explicitly import the Tonic framework are searched for top-level blueprints. This approach offers a course-grained control over the blueprint generation process. For example, none of the iTasks core modules import the Tonic framework, so no core tasks are turned to blueprint.

All blueprints are built from a small and *general* core language, shown below. At compile-time, we generate blueprints per Clean module (`TModule`). For every function of a blueprint monad we create a `TFun` record. This record contains meta-information, such as the comments, module name, function name, the line number of the function definition, the result-type, the argument names and types and the function body. Every type or expression is represented by the `TExpr` data type.

```
:: ModuleName  :== String
:: FuncName    :== String
:: Pattern     :== TExpr
:: TypeName    :== String
:: PPExpr      :== String
:: ExprId      :== [Int]
:: VarName     :== String
:: VarPtr      :== Int

:: TModule   = { tm_name  :: ModuleName
               , tm_funcs :: Map FuncName TFun }

:: TFun      = { tf_comments  :: String
               , tf_module    :: ModuleName
               , tf_name      :: FuncName
               , tf_iclLineNo :: Int
               , tf_resty     :: TExpr
               , tf_args      :: [(TExpr, TExpr)]
               , tf_body      :: TExpr }

:: TExpr     = TVar    ExprId PPExpr VarPtr
             | TPPExpr PPExpr
             | TMApp   ExprId (Maybe TypeName) ModuleName
                       FuncName [TExpr] TPriority (Maybe VarPtr)
             | TFApp   ExprId FuncName [TExpr] TPriority
             | TLam    [TExpr] TExpr
             | TLet    [(Pattern, TExpr)] TExpr
             | TIf     ExprId TExpr TExpr TExpr
             | TCase   ExprId TExpr [(Pattern, TExpr)]

:: TPriority = TPrio TAssoc Int | TNoPrio

:: TAssoc    = TLeftAssoc | TRightAssoc | TNoAssoc
```

`TExpr` contains the usual suspects for a small core language, such as variables, literals, lambdas, **let**s and **case**s. Function application, however, is represented by two distinct constructors: `TMApp` and `TFApp`. The former represents function application of all contained monads (hence the M), the latter all other function applications. Several constructors contain additional meta-data. An `ExprId`, found in the `TVar`, `TMApp`,

TFApp, TIf, and TCase constructors, uniquely identifies those expressions in a blueprint. This turns out to be very useful later on when we will make blueprints show dynamic behaviour (Section 4). TMApp also contains the type of the monad (if the fuction is monomorphic in its monadic return type) and the name of the module in which the function being applied is defined. This is to disambiguate functions with the same name. In addition to the function's arguments and priority, it has an optional VarPtr in case the function being applied is variable.

4. DYNAMIC BLUEPRINTS

A static blueprint gives a graphical view of how the monad combinators are defined in the source code. Now we want be able to trace and inspect the execution of the resulting application, making use of the static blueprints. Although the monad parts of the program may be just a small part of the source code, they are an important part and they commonly form the backbone of the architecture of the application. If we can follow their execution and see how their corresponding blueprints are being applied, we will already have a good impression of the run-time behaviour of the application. We want to show which monadic computation is currently being executed, how far along the program's flow we currently are, the current value for a given argument or variable, the result of a completed computation, and which program branches will be taken in the future. Before delving into the technical challenges associated with addressing these requirements, lets look at our previous examples and how their static blueprints are used at run-time.

When a function with a Blueprint-monadic type is applied, we make an instantiation (a copy) of its corresponding static blueprint, creating a *dynamic blueprint*. On top of it we can show who is calling it, we can inspect its actual arguments, and visualize the progress in the flow when the body is being executed. The Tonic viewer can show and inspect these dynamic blueprints. Notice that the Tonic viewer can show the blueprints in real-time, i.e. when the application is being executed. The Tonic viewer also allows inspecting the past, and it can sometimes predict the future.

4.1 Dynamic Blueprints of the Task Monad

In this section we will look at how we augment the blueprints of the previous examples with run-time information.

4.1.1 Prime Number Checker

In the primeCheck example we saw sequential composition using a bind combinator. Since bind determines the order in which computations are executed, it is a great place for us to track progress in a program's flow. Figure 8 shows the dynamic blueprints for the primeCheck iTasks program as it is executed.

When the program starts and the user is presented with the input field, its corresponding blueprint instance is that of Figure 8(**a**). Immediately the blueprint is different from its static incarnation in several ways. A pair of numbers is added in the top bar, next to the task name. This is the *task ID*, uniquely identifying this task instance within the iTasks run-time system. Next to it is the image of a person, together with the name of the person that is currently executing this particular task instance. Going to the lower half of the blueprint, we see that the upper area of the task-application node is coloured green. Green means that the task is currently actively being worked on. We also say that the enterNumber node is *active*. Additionally, the task ID of the enterNumber task instance is added to the blueprint and positioned next to the task name.

Next to each node, a square is drawn. Clicking on this square allows us to inspect the task's value in real-time. Its colour also indicates the stability of the task's value. In Figure 8(**a**), there is no value yet, hence the square is white. This is confirmed by a pop-up window when we click the white square. However, as soon as a number is entered by the end user in the editor's text field, or whenever the number is changed, the current input is directly shown in the inspection window (Figure 8(**b**)).

On the right side of the blueprint there is a diamond-shaped conditional node, followed by two nodes labeled with viewInformation, which now have green borders. These border colours tell us something about the future, in particular which program branch might be taken. Since the program has only just started, all branches might still be reached. However, when we enter the number 42 to the enterNumber task's text field – which is not a prime number – we can already predict that the True branch will not be reached. This is represented by red borders, as seen in Figure 8(**b**). If we changed the number in the box to, e.g., 7 the tasks in the False branch would receive a red border instead. We call this feature *dynamic branch prediction*. Once the user has entered a number and has pressed "Continue", the workflow progresses to the second task and the blueprint instance is updated accordingly (Figure 8(**c**)). The first node is no longer highlighted. Instead, it is *frozen* and given a blue colour. A frozen blueprint node for a given task instance will not change again. Additionally, the edge between the first and second node is now coloured green. For edges, green does not indicate activity, but the *stability* of the previous task's value. A green edge means an unstable value, while a blue edges means a stable value. In iTasks, tasks may have a stable or unstable value, or even no value at all. It reflects the behaviour of an end user filling in a form. The form may be empty to start with or some information may be entered which can be changed over time. Once values are stable they can no longer change over time. When the "Continue" button is pressed again, we reach the False branch, as predicated earlier. Since the True branch is no longer reachable, its nodes now get a grey header (Figure 8(**d**)).

4.1.2 Recursion and Higher-Order Tasks

Yet other dynamic behaviour is found in the blueprints of add1by1 (as applied in addPalindromes), in which we have to deal with a task as argument, a step combinator, and recursion. Its dynamic blueprints are shown in Figure 9. Notice that the task variable is now replaced by a task-application node containing the name of the palindrome task (Figure 9(**a**)). When a valid palindrome has been entered, the workflow continues to the viewInformation task. The step combinator at that point presents the user with two buttons: "Add another" and "Done". The former can always be pressed, whereas the latter is only enabled when at least two values are accumulated in newList. Since we only have one palindrome so far, only the "Add another" button is enabled. This is reflected in the blueprint (Figure 9(**b**)). Recursion is simply yet another task-application node (Figure 9(**c**)). Entering the recursion creates a new blueprint instance for the add1by1 task in which another palindrome task is executed

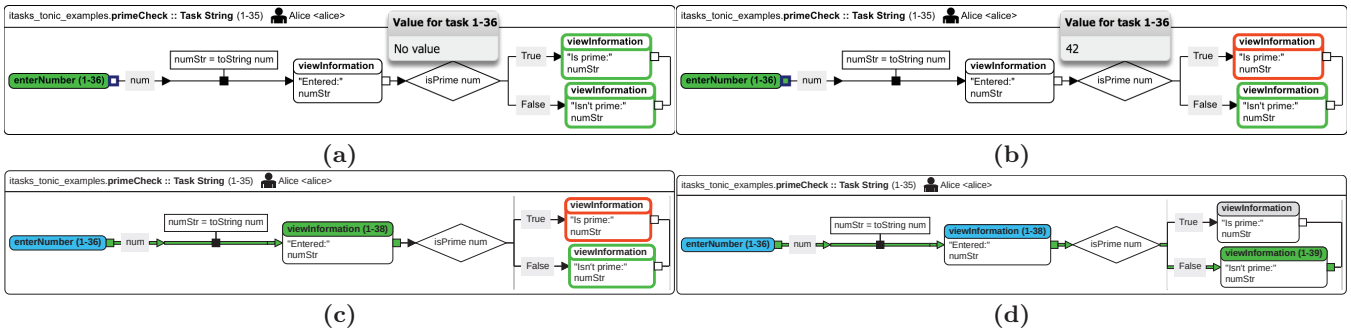

Figure 8: Dynamic blueprints of `primeCheck` showing monadic progress tracking and value inspection

(Figure 9(**d**)). When the user submits another valid palindrome, we encounter `viewInformation` again. This time, however, the "Done" button is enabled, because the `hasManyElems` predicate holds. (Figure 9(**e**)). Pressing "Done" finishes the `add1by1` task and returns the list of palindromes (Figure 9(**f**)).

4.1.3 *Parallel Chat Tasks*

In the `parallelChat` example we saw that the function application of `chatTasks` can only be pretty printed. There are two reasons for this: 1) we don't have a `Contained` instance for lists (for the sake of this example), and 2) `chatTasks` is a function application. We cannot compute any kind of function statically. At run-time, however, we would like to know which tasks are being executed in parallel, so we need to replace the pretty-printed expression with a list of task-application nodes dynamically. We can see how Tonic deals with this situation in Figure 10.

Figure 10(**a**) shows that we select two friends to chat with: Bob and Carol. Next, the `parallelChat` task delegates three chat tasks: one to the current user, Alice, and one to each of her friends. Since the `chatTasks` function application is now evaluated, we can substitute a list of task application nodes for the pretty-printed expression. Each of the nodes contain the parallel task's name and task ID. For each of these nodes a corresponding blueprint instance is created, which can be inspected as well (Figure 10(**a**)).

4.2 Tonic Architecture

To enable such dynamic features, we need to make a connection between the static blueprints and the program's run-time. With this connection, we can pass additional information from the original program to the Tonic run-time system. This is similar to standard tracing and debugging tools. Connecting blueprints and a program's run-time is done by extending the `Contained` and `Blueprint` classes with *wrapper functions* that we apply to the original program at compile-time. These wrapper functions are executed at the same time as the program's original functions. It is up to the programmer to provide sensible instances for these classes. We have already provided instances for both classes for the `Task` type that can be used in any iTasks program. Section 4.4 will show how these classes are defined for iTasks.

Figure 11 shows the architecture of a Tonic-enabled iTasks application. Not all functions are augmented with wrapper functions. Only monads for which `Blueprint` or `Contained` instances exist are turned to blueprints. Even if such instances exist, this only happens when the module that these functions inhabit imports the Tonic framework. This mechanism serves as a coarse-grained filter. For example, none of the iTasks core modules import the Tonic framework, hence none of the iTasks core modules are transformed by the Clean-Tonic compiler. Tonic also has a fine-grained filter to regulate wrapping behaviour on the function-application level. More on this in Section 4.2.2.

Tonic maintains an SDS with run-time information. When a wrapper function is applied, it writes additional information to this SDS, allowing us to track the program's progress and inspect its values. The specifics of what data the wrappers contain are discussed in Section 4.2.2 and Section 4.2.3. Writing to the share triggers an update that refreshes the dynamic blueprint.

4.2.1 *The Tonic Viewer*

The Tonic viewer is written in iTasks itself and is simply yet another task. Using the Tonic viewer in an iTasks application simply requires the programmer to make sure the viewer task is reachable by the program's end user. Section 4.5 talks about a solution that does not have this requirement. Having the viewer built into the application that is going to be visualized does have certain advantages. In iTasks' particular case, this allows us to easily inspect nearly all function arguments and task values using iTasks' own generic editors. This even works for complex types.

When applying the viewer-task, the programmer can optionally provide additional render functions with which the rendering of individual function-application nodes can be customized. The programmer can use our fully declarative SVG library [1] to define alternative visualizations.

4.2.2 *Contained Monads*

The `Contained` class is what identifies interesting function applications. It is therefore the right place to gather more information about the functions being applied. For example, which function is being applied? To which blueprint node does this function application correspond? How does the value of the underlying function application influence the program's workflow? We extend the `Contained` class with only one function: `wrapFunApp`, as shown below.

```
class Contained m | Monad m where
  wrapFunApp :: ModuleName FuncName ExprId [(ExprId, a -> Int)]
                (m a) -> m a | iTask a
```

It is here that the Tonic system is notified of the execution of individual computations, where the current value of these

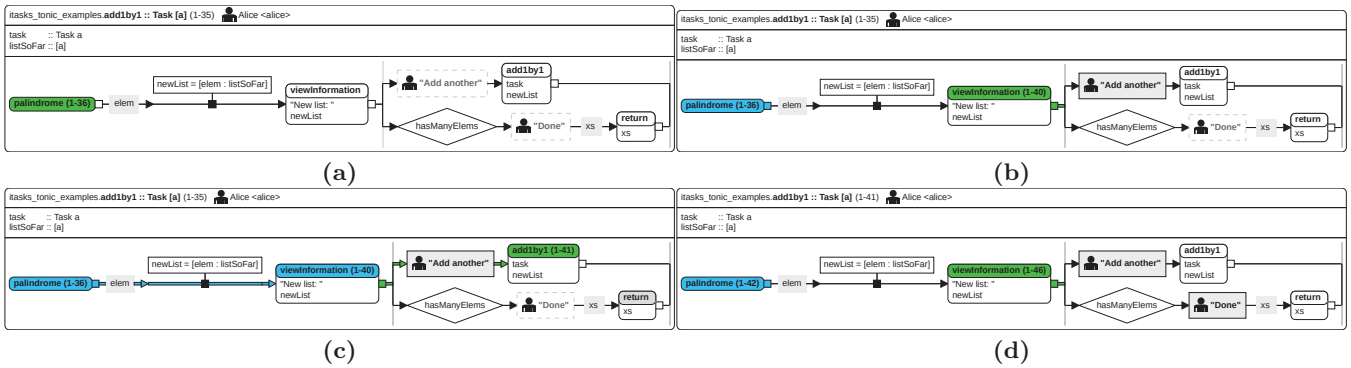

Figure 9: Dynamic blueprints of `add1by1`

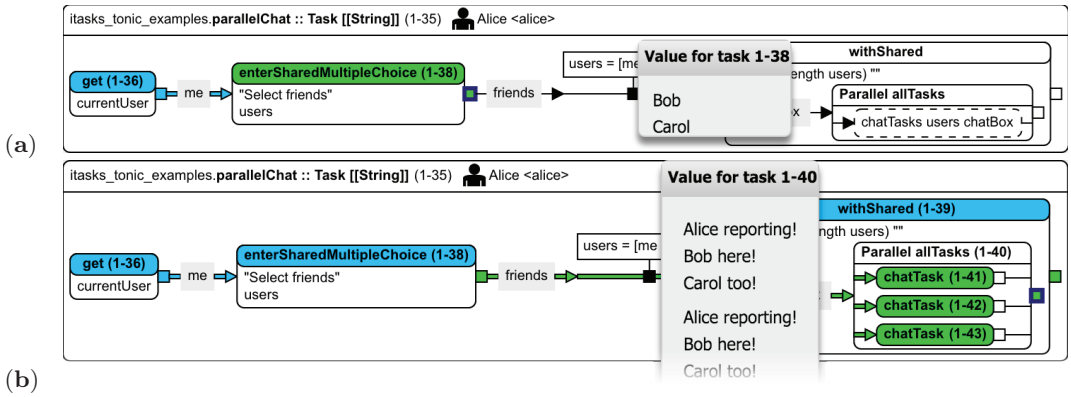

Figure 10: Dynamic blueprints for the parallel chat example

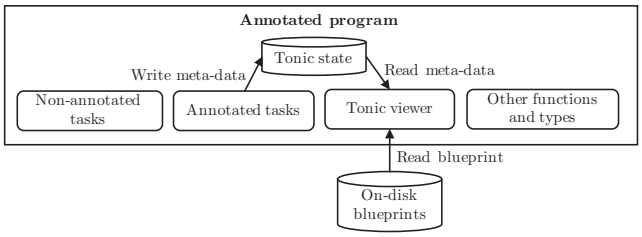

Figure 11: Tonic architecture

computations is inspected, where blueprints are updated dynamically, et cetera. `wrapFunApp` takes five arguments. The last argument is the original computation. Clean's strictness analyser ensures that this argument will have the same strictness (or laziness) as the wrapped computation. This implies that `wrapFunApp` in its most basic definition is simply the identity function. The first two arguments of `wrapFunApp` are the module and function name of the function being applied. The third argument is an `ExprId`, which, together with the module and function name of the function application's context (obtained via the `Blueprint` class; see also Section 4.2.3), uniquely identifies this function application. The same `ExprId` is also found in the blueprint of the parent function, allowing us to relate run-time execution to the static blueprint. The fourth argument allows us to do dynamic branch prediction. It is a list of pairs, the first element of which is the `ExprId` that refers to the **case** block of which we want to predict its future. The second element is a function that, given the value of type `a` of the computation `(m a)` being wrapped, gives the index number of the branch that will be chosen, should that value be used. Before discussing how the `Contained` class is used we need to understand how dynamic branch prediction works.

Tonic's dynamic branch prediction feature utilizes the fact that Tonic is implemented as a compiler pass in the Clean compiler. During the Tonic pass, we *copy* **case** blocks and lift them to a newly generated function. We transform the right-hand side of the individual cases and return the *index* of the branch as integer. We call this procedure *case cloning*. By applying this fresh function, we known, using the original case expression, the index of the branch that will be taken, should that expression be evaluated with an identical context. Definition 4.1 formalizes this process.

Definition 4.1. Case cloning transformation. Given a **case** expression

$$\begin{aligned} &\textbf{case } f\ x_1 \ldots x_i \textbf{ of} \\ &\quad p_1 \rightarrow e_1 \\ &\quad \ldots \\ &\quad p_j \rightarrow e_j \end{aligned}$$

Generate a fresh function

$$dbpf :: a_1 \ldots a_i \to Int$$
$$dbpf\ x_1 \ldots x_i = \textbf{case}\ f\ x_1 \ldots x_i\ \textbf{of}$$
$$p_1 \to 1$$
$$\ldots$$
$$p_j \to j$$

As mentioned earlier, wrappers are not applied all the time. In particular, it might be necessary to forego wrapping certain expressions when they are argument to another function. Consider again the `add1by1` example. Should we wrap the recursive call as well as the `task` variable, the recursive instance would effectively have two wrappers around `task` due to laziness. When `task` is evaluated, both wrappers would be evaluated as well, polluting Tonic's run-time state with wrong data. Still, in some cases we do want to wrap higher-order arguments. The most prominent case for this is the bind combinator. An iTasks-specific case are the parallel combinators. They are rendered as containers within which we want to keep following the workflow's progress. We need the wrappers to do so. To solve this problem, we only wrap function arguments when the function itself comes from a module that does not enable Tonic. In addition, a function-level pragma, either `TONIC_CONTEXT` or `TONIC_NO_CONTEXT`, can be provided. When the former pragma is used, the function's arguments are wrapped. With the latter, they are not. The pragmas override the default module-based wrapping behaviour. Definition 4.2 formalizes the transformations the Tonic compiler applies to utilize the `Contained` class.

Definition 4.2. `Contained` transformation. For all function applications $f e_1 \ldots e_i$ where $f :: \alpha_1 \ldots \alpha_i \to m\ \alpha_n$ and f is in module M, and for which holds **instance** `Contained` m:

$[\![f e_1 \ldots e_i \texttt{ >>= } \lambda v.e_j]\!]$
iff module `M` does not enable Tonic, or
f has `TONIC_CONTEXT`
$\Rightarrow$
`wrapFunApp` "`M`" "`f`" $exprId(f)$ $dbpC(v, e_j)$
(f $[\![e_1]\!]$ ... $[\![e_i]\!]$) `>>=` $\lambda x.[\![e_j]\!]$

$[\![f e_1 \ldots e_i \texttt{ >>= } \lambda v.e_j]\!]$
otherwise
$\Rightarrow$
`wrapFunApp` "`M`" "`f`" $exprId(f)$ $dbpC(v, e_j)$ $(f e_1 \ldots e_i)$
`>>=` $\lambda x.[\![e_j]\!]$

$[\![f e_1 \ldots e_i]\!]$
iff module `M` does not enable Tonic, or
f has `TONIC_CONTEXT`
$\Rightarrow$
`wrapFunApp` "`M`" "`f`" $exprId(f)$ `[]` $(f [\![e_1]\!] \ldots [\![e_i]\!])$

$[\![f e_1 \ldots e_i]\!]$
otherwise
$\Rightarrow$
`wrapFunApp` "`M`" "`f`" $exprId(f)$ `[]` $(f e_1 \ldots e_i)$

$[\![e]\!]$
$\Rightarrow$
e

Two additional functions are used during this transformation: *exprId* and *dbpC*. $exprId(f)$ returns a unique identifier for the application of f to its arguments. *dbpC* enables dynamic branch prediction for contained monads as follows. For all lifted case functions $dbpf_k\ x_1 \ldots x_i, x_{i+1}$ from e_j, if $v \equiv x_{i+1}$ and $x_1 \ldots x_i$ are bound, then $[(caseExprId(dbpf_1), dbpf_1\ x_1 \ldots x_i), \ldots, (caseExprId(dbpf_n), dbpf_n\ x_1 \ldots x_i)]$. Here, *caseExprId* returns the unique identifier for the original case expression that was used to create *dbpf*. Implementing dynamic branch prediction in a bind is possible because the monad right-identity law guarantees that for any expression e_1 `>>=` $\lambda x.e_2$, x will always bind e_1's result value.

4.2.3 Blueprint Monads

The `Blueprint` class already allows us to identify functions for which to generate a blueprint. This class is therefore well suited to capture some meta data for blueprint functions that would otherwise be lost at run-time. We extend the `Blueprint` class with two functions: `wrapFunBody` and `wrapFunArg`, as shown below.

```
class Blueprint m | Contained m where
  wrapFunBody :: ModuleName FuncName [(VarName, m ())]
                 [(ExprId, Int)] (m a) -> m a | iTask a
  wrapFunArg  :: VarName a -> m () | iTask a
```

The `wrapFunBody` function is statically applied to the body of a blueprint function. It has several goals: to make the blueprint function's module and function name available at run-time, to provide a way to inspect the blueprint function's arguments, and to do future branch prediction based on the function's arguments. The `wrapFunArg` function is used in the third argument of `wrapFunBody`. It is statically applied to all function arguments to enable their inspection at run-time. In general, the compiler applies following transformation rule:

Definition 4.3. `Blueprint` transformation. For all function definitions $f :: \alpha_1 \ldots \alpha_i \to m\ \alpha_n$ in module M, for which holds **instance** `Blueprint` m:

$[\![f x_1 \ldots x_i = e]\!]$
$\Rightarrow$
$f\ x_1 \ldots x_i =$ `wrapFunBody` "`M`" "`f`"
[("x_1", `wrapFunArg` x_1)
, ...
, ("x_i", `wrapFunArg` x_i)]
$dbpB(x_1 \ldots x_i, e)$
$[\![e]\!]$

dbpB works subtly different from *dbpC*. Rather than being associated with a variable bound by a lambda in a bind, it works on the function's arguments, which are all bound as soon as the blueprint is instantiated.

The `iTask` constraint on the `Contained` and `Blueprint` class members is used extensively in iTasks. Unfortunately, due to limitations in Clean's type system, we are currently forced to include this context restriction in our two classes, even though they might be instantiated for monads that have nothing to do with iTasks. We will come back to this limitation in Section 6.

4.3 Tonic Wrappers in Action

Applying all transformations to the `primeCheck` example transforms it to the following code (slightly simplified for readability). Module names passed to the wrappers are fully qualified. The lists of numbers are the unique expression identifiers from the *exprId* function. The `_f_case` function is an instance of the *dbpf* function.

```
primeCheck :: Task String
primeCheck
  = wrapFunBody "itasks_tonic_examples" "primeCheck" [] []
    wrapFunApp "itasks_tonic_examples" "enterNumber" [0, 0]
      [([0, 1, 0, 1, 0], _f_case_4566316320)] enterNumber
  >>= \num. let numStr = toString num in
    wrapFunApp "iTasks.API.Common.InteractionTasks"
      "viewInformation" [0, 1, 0, 0] []
      (viewInformation "Entered:" numStr)
  >>| if (isPrime num)
        (wrapFunApp "iTasks.API.Common.InteractionTasks"
          "viewInformation" [0, 1, 0, 1, 0, 0] []
          (viewInformation "Is␣prime:" numStr)
        (wrapFunApp "iTasks.API.Common.InteractionTasks"
          "viewInformation" [0, 1, 0, 1, 0, 1] []
          (viewInformation "Isn't␣prime:" numStr)
```

4.4 Dynamic Blueprints in iTasks

To demonstrate how these wrappers can be used, we show their concrete implementation for iTasks. A task in iTasks is represented by the `Task` type as follows:

```
:: Task a = Task (TaskAdministration TonicAdministration
                  *IWorld -> *(TaskResult a, *IWorld))
```

A task is implemented as a continuation which takes some internal task-administration and some Tonic administration and passes it down the continuation. It chains a unique `IWorld` through the continuation, which allows interaction with SDSs and provides general IO capabilities, amongst other things. The continuation produces a `TaskResult`, which, amongst other things, contains the task's result value.

The wrappers we place in the code unpack the continuation from a `Task` constructor and use it to define a new task. In the case of the `Blueprint` class, the wrapper's job is to create a new blueprint instance for the task that is being started (line 8), while in the case of `Contained`, the wrapper's job is to update the blueprint instance in which the task-application takes place. In that case, a blueprint instance already exists and just needs to be loaded from Tonic's internal administration (line 27). Both wrapper classes perform similar operations: the relevant blueprint instance is loaded and updated, after which it is stored again, triggering a redraw event. One of the differences between the two classes is in when the original continuation is executed. In `wrapFunBody`, it is the last thing the wrapper does (line 14). In `wrapFunApp`, the original continuation is executed half-way in the wrapper (line 34). After executing the original continuation, the blueprint instance is loaded again, since it may have been updated by other wrappers in the mean time. Another thing both class instances have in common is that they both do future branch prediction (lines 10 and 38).

```
1  instance Blueprint Task where
2    wrapFunBody :: ModuleName FuncName [(VarName, Task ())]
3                   [(ExprId, Int)] (Task a) -> Task a | iTask a
4    wrapFunBody modNm funNm args dbp (Task oldEval)
5      = Task newEval
6      where
7      newEval taskAdmin tonicAdmin iworld
8        # (blueprint, iworld) = instantiateBlueprint modNm
9                                  funNm taskAdmin args iworld
10       # blueprint           = processCases dbp blueprint
11       # iworld              = storeBlueprint blueprint iworld
12       # tonicAdmin          = updateTonicAdminFun blueprint
13                                 tonicAdmin
14       = oldEval taskAdmin tonicAdmin iworld
15
16   wrapFunArg :: String a -> Task () | iTask a
17   wrapFunArg descr val = viewInformation descr val @! ()
18
19 instance Contained Task where
20   wrapFunApp :: ModuleName FuncName ExprId
21                 [(ExprId, a -> Int)] (Task a)
22              -> Task a | iTask a
23   wrapFunApp modNm funNm exprId dbp (Task oldEval)
24     = Task newEval
25     where
26     newEval taskAdmin tonicAdmin iworld
27       # (blueprint, iworld) = getBlueprintInstance tonicAdmin
28                                 iworld
29       # (blueprint, iworld) = preEvalUpdate modNm funNm
30                                 exprId blueprint iworld
31       # iworld              = storeBlueprint blueprint iworld
32       # tonicAdmin          = updateTonicAdminApp modNm funNm
33                                 tonicAdmin exprId
34       # (result, iworld)    = oldEval taskAdmin tonicAdmin
35                                 iworld
36       # (blueprint, iworld) = getBlueprintInstance tonicAdmin
37                                 iworld
38       # blueprint           = processCases dbp result
39                                 blueprint
40       # (blueprint, iworld) = postEvalUpdate result modNm
41                                 funNm exprId blueprint iworld
42       # iworld              = storeBlueprint blueprint iworld
43       = (result, iworld)
```

4.5 Dynamic Blueprints of the I/O Monad

We claim that Tonic supports any monad, so lets look at how Tonic handles the `IO` variant of the `primeCheck` example. Figure 12 shows a dynamic blueprint for `primeCheck` at the point where the user has to enter a number. This dynamic blueprint is produced by an experimental stand-alone Tonic viewer, which serves purely as a proof-of-concept. Instead of including the Tonic viewer as part of an iTasks program, this version of the viewer receives dynamic updates via a TCP protocol. Figure 13 shows the architecture of this stand-alone Tonic viewer. The current implementation has several limitations. It is not possible to inspect values or do dynamic branch prediction, nor is it possible to select which blueprint instance you are interested in; the viewer always ever shows the blueprint instance for which the latest update arrived. Still, we feel like this is an important step towards positioning Tonic as a general tool.

5. RELATED WORK

Tonic can be seen as a graphical tracer/debugger. Several attempts at tracer/debuggers for lazy functional languages have already been made. Some examples include Freja [6, 5], Hat [16, 17], and Hood [3], the latter of which also has a graphical front-end called GHood [15]. All of these systems are general-purpose and in principle allow debugging of any functional program at a fine-grained level. Tonic only allows tracing on a monadic abstraction level. Due to our focus on monads, Tonic does support any monad, including the IO monad. All of the aforementioned systems only have limited support for the IO monad. Freja is implemented as its own compiler which supports a subset of Haskell 98. Previous Hat versions were implemented as a part of the nhc98 compiler. Modern Hat versions are implemented as stand-alone programs and support only Haskell 98 and some bits of Haskell 2010. Tonic is implemented in the Clean compiler and supports the full Clean language,

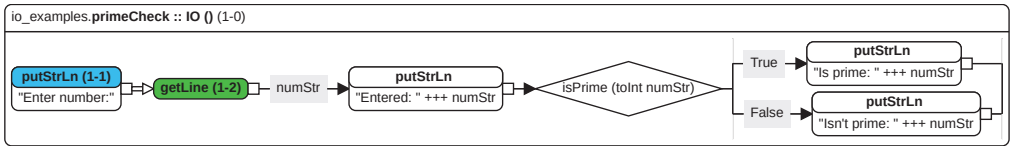

Figure 12: Dynamic blueprint for `IO` variant of `primeCheck`

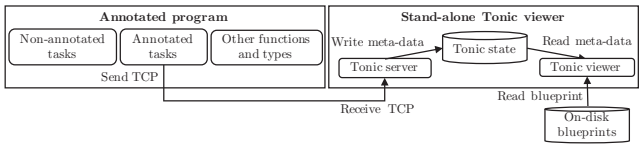

Figure 13: Architecture of the stand-alone Tonic viewer

which is more expressive than even Haskell 2010. Hood, on the other hand, requires manually annotating your program with trace functions.

GHood is a graphical system on top of Hood that visualizes Hood's output. Its visualizations are mostly aimed at technical users. Graphical programming language, such as VisaVis [13] and Visual Haskell [14] suffer from similar problems. Tonic explicitly aims at understandability by laymen by choosing a higher level of abstraction, hiding details that do not contribute significantly to understanding the program, and by utilizing coding conventions.

Tonic can also be seen as a graphical communication tool. In this sense it fulfils a role similar to UML [8, 9] and BPMN [19]. Both of these technologies also offer a means to *specify* programs and workflows. This is something Tonic is *not* designed to do. Previous work from our group, GiN – Graphical iTasks Notation [4] can be used for that.

6. DISCUSSION AND CONCLUSION

In this paper we generalised and expanded our original Tonic idea. Any monadic program can now be statically visualized by Tonic. While dynamic visualization is currently limited to iTasks, we have laid the foundation for dynamically visualizing any monadic program.

So far, we have extensively experimented with using Tonic for iTasks. Our approach of using type classes for defining how dynamic behaviour should be captured allows for an almost completely orthogonal implementation for iTasks; the core system only required very minimal changes. The biggest change was made to the way iTasks handles task IDs. These IDs are not generated deterministically, so we had to implement a form of stack-tracing in iTasks to capture which tasks had already been executed. Systems with deterministic identifiers will not have to resort to such measures. The minimal implementation for the IO monad is completely orthogonal. This suggests that Tonic can be used in other contexts as well, as illustrated with the proof-of-concept implementation for the `IO` monad (Section 4.5).

6.1 Stand-alone Dynamic Tonic Viewer

Section 4 showed a proof-of-concept Tonic viewer for the `IO` monad. While it works reasonably well for simple programs, it lacks features and is not very user-friendly. In the future we want to expand this stand-alone viewer to the point where it can replace the built-in iTasks Tonic viewer.

6.2 Complete iTasks Agnosticism

Even though we have made the Tonic compiler completely iTasks-agnostic, Tonic itself still is tied to iTasks by means of the `iTask` context restriction in the `Blueprint` and `Contained` classes. The `iTask` class is used to be able to generically inspect values. Its presence in the classes means that, even when using Tonic for non-iTasks programs, we require an iTasks-specific class to be instantiated for all types that we want to inspect. Clean's type-system, however, offers no elegant solution to this problem. GHC in particular could solve this problem elegantly using its `ConstraintKinds` and `TypeFamilies` extensions, as shown in the code snippet below. Here, the context restriction depends on the type of the `Blueprint` monad.

```
class Monad m => Contained m where
  type CCtxt m a :: Constraint
  type CCtxt m a = ()
  wrapFunApp  :: CCtxt m a
              => (ModuleName, FuncName) -> ExprId -> m a -> m a

class Contained m => Blueprint m where
  type BpCtxt m a :: Constraint
  type BpCtxt m a = ()
  wrapFunBody :: BpCtxt m a
              => ModuleName -> FuncName -> [(VarName, m ())]
              -> m a -> m a
  wrapFunArg  :: BpCtxt m a => String -> a -> m ()

instance Contained Task where
  type CCtxt Task a = ITask a
  wrapFunApp  = ..

instance Blueprint Task where
  type BpCtxt Task a = ITask a
  wrapFunBody = ..
  wrapFunArg  = ..

instance Contained Maybe where
  wrapFunApp  = ..
```

For Clean we could compromise by, e.g., requiring all values to be serializable to a string. While this approach would generalise the Tonic classes in the short term, it limits the ways in which we can present the inspected values. For example, using the built-in iTasks Tonic viewer, we can currently render fully interactive graphics, such as a Google Maps widget, in the Tonic inspector. A true solution would be to implement variable context restrictions in type classes in Clean, similar to GHC.

6.3 Portability

By generalising Tonic it becomes clear that it could be implemented in a context different from Clean as well. Ac-

knowledging that GHC in particular offers elegant solutions to improve Tonic's type classes, it would be interesting to explore porting Tonic to GHC.

6.4 Dynamic Blueprint Modification

Tonic's blueprints, whether static or dynamic, are currently read-only. We cannot influence the execution of programs or change a program's implementation. In the future we would like to explore such possibilities.

6.5 Wrapping up

Tonic, as presented in this paper, lays the foundation for a plethora of distinct but related tools. On the one hand, blueprints can be seen as automatic program documentation. Each time the program is compiled, its blueprints are generated too, giving the programmer up-to-date documentation for free. Furthermore, dynamic instances of these blueprints document the program's dynamic behaviour. Due to the blueprint's high level of abstraction, this free documentation can serve as the basis of communication between various project stakeholders as well, enabling rapid software development cycles. Whether Tonic succeeds in being a suitable communication tool is a subject for future work.

Another way to look at Tonic is as a graphical tracer and debugger. Dynamic blueprints trace the execution of the program, while Tonic's inspection and future branch prediction capabilities add features desirable in a debugger. Even for programmers, having such information visualized may aid in understanding the programs better, and in constructing the required program faster or with less effort.

Yet another avenue worth exploring is education. We are currently including blueprints in the lecture slides of functional programming courses. In our experience, students struggle with the concept of monads, so we want to see if and how Tonic can reduce these problems.

Particularly if we continue working on a stand-alone Tonic viewer, we are certain that many additional practical use-cases for Tonic will be be discovered.

7. ACKNOWLEDGMENTS

This research is funded by the Royal Netherlands Navy and TNO. We thank all the reviewers for their constructive and useful feedback.

8. REFERENCES

[1] P. Achten, J. Stutterheim, L. Domoszlai, and R. Plasmeijer. Task oriented programming with purely compositional interactive scalable vector graphics. In S. Tobin-Hochstadt, editor, *Proceedings of the 26Nd 2014 International Symposium on Implementation and Application of Functional Languages*, IFL '14, pages 7:1–7:13, New York, NY, USA, 2014. ACM.

[2] E. Arisholm, L. C. Briand, S. E. Hove, and Y. Labiche. The impact of UML documentation on software maintenance: An experimental evaluation. *Ieee Transactions on Software Engineering*, 32(6):365–381, June 2006.

[3] A. Gill. Debugging Haskell by Observing Intermediate Data Structures. *Electr Notes Theor Comput Sci*, 2000.

[4] J. Henrix, R. Plasmeijer, and P. Achten. GiN: A Graphical Language and Tool for Defining iTask Workflows. *Trends in Functional Programming*, pages 163–178, 2012.

[5] H. Nilsson, U. i. L. D. o. C. Science, and Information. Declarative Debugging for Lazy Functional Languages, 1998.

[6] H. Nilsson and J. Sparud. The Evaluation Dependence Tree as a Basis for Lazy Functional Debugging. *Automated software engineering*, 4(2):121–150, 1997.

[7] Object Management Group. Business process model and notation (BPMN) version 1.2. Technical report, Object Management Group, 2009.

[8] Object Modeling Group. OMG Unified Modeling Language Specification. Technical report, Mar. 2000.

[9] Object Modeling Group. OMG Unified Modeling Language (OMG UML), Infrastructure. Technical report, Mar. 2012.

[10] OMG. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1, August 2011.

[11] R. Plasmeijer, B. Lijnse, S. Michels, P. Achten, and P. Koopman. Task-Oriented Programming in a Pure Functional Language. In *Proceedings of the 2012 ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP '12*, pages 195–206, Leuven, Belgium, Sept. 2012. ACM.

[12] R. Plasmeijer and M. van Eekelen. Clean language report (version 2.1). http://clean.cs.ru.nl, 2002.

[13] J. Poswig, G. Vrankar, and C. Morara. VisaVis: a Higher-order Functional Visual Programming Language. *Journal of Visual Languages & Computing*, 5(1):83–111, Mar. 1994.

[14] H. J. Reekie. Visual Haskell: a first attempt. Technical report, 1994.

[15] C. Reinke. GHood–graphical visualisation and animation of Haskell object observations. *2001 ACM SIGPLAN*, 2001.

[16] J. Sparud and C. Runciman. Complete and partial redex trails of functional computations. In *Implementation of Functional Languages*, pages 160–177. Springer Berlin Heidelberg, Sept. 1997.

[17] J. Sparud and C. Runciman. Tracing lazy functional computations using redex trails. *Programming Languages: Implementations*, 1997.

[18] J. Stutterheim, R. Plasmeijer, and P. Achten. Tonic: An Infrastructure to Graphically Represent the Definition and Behaviour of Tasks. In J. Hage and J. McCarthy, editors, *Trends in Functional Programming*, volume 8843 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2014.

[19] S. A. White. Business Process Model and Notation, V1.1. pages 1–318, Jan. 2008.