Gin: Graphical iTask Notation – extended abstract –

Jeroen Henrix, Rinus Plasmeijer, and Peter Achten

Institute for Computing and Information Sciences
Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
jmhenrix@student.ru.nl, {rinus, p.achten}@cs.ru.nl

1 Introduction

In this paper we present Gin. Gin is both a graphical notation for the iTask system as well as a tool to construct iTask applications in an interactive and graphical way. The iTask system [3] is a workflow management system (WFMS) that is embedded in the pure and lazy functional programming language Clean. WFMSs are software applications that coordinate business processes. This coordination is based on a workflow model: a formal description in a workflow definition language (WDL) of the tasks that comprise a business process, their ordering and data dependencies.

Traditionally, WDLs are based on coloured Petri Nets [1]. This has the immediate advantage that they come with an intuitive graphical notation that can be used in the development process by workflow engineers and domain experts. The iTask WDL, on the other hand, is founded on a set of combinator functions, which is a common approach in the functional language community. Hence, to understand and appreciate an iTask workflow, one needs to be trained in functional programming. Using Gin, this is less of a requirement, because Gin offers a structured and graphical way of building iTask models, and confronts the user much less with functional programming activities. Hence, building a simple model should also be simple to the user, whereas for the development of a complex model one is likely to be better off using the full expressive power of Clean.

The contributions of our work are:

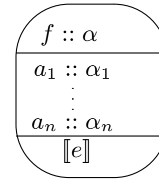
- We identify graphical workflow patterns that correspond with iTask combinators. We adopt graphical conventions from the standard workflow community if possible. We integrate combinator language features (scoping of variables, recursion, block structure) within Gin.
- We implement the Gin tool as a proof of concept. It is integrated in the iTask system and uses the Clean compiler for error and type checking. Errors are reported back to the user in terms of the Gin model.

In the remainder of this extended abstract we briefly discuss the Gin language (Section 2) and tool (Section 3). We end with conclusions (Section 4). *Related work will be discussed in the full paper.*

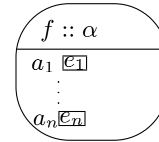
2 The Gin language

The Gin language is a mostly graphical WDL to which Clean text fragments can be added. It defines a graphical notation for a subset of the iTask WDL. iTask expressions of type *Task* α are represented in Gin by directed graphs. As conventional in graphical WDLs, nodes indicate tasks, and edges indicate control flow relations. Because the iTask combinators are block structured, unstructured control flow like arbitrary jumps cannot be expressed in iTask. Therefore, Gin only allows well-structured flows, consisting of atomic tasks and nested, non-overlapping control blocks. We informally introduce the Gin language by means of a map $[\![\cdot]\!]$ from iTask expressions to Gin diagrams. Gin supports the following constructs:

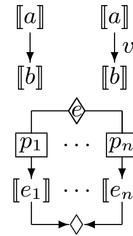
- *Task definition*: Functions $f :: \alpha_1 \rightarrow \alpha_2 \rightarrow \ldots \alpha_n \rightarrow \mathit{Task}\ \alpha$ $(n \geq 0)$ defined as $f\ a_1\ a_2 \ldots a_n\ =\ e$ are depicted as shown on the right. The arguments a_i are simple variable names, patterns are not supported. The variables a_i are in scope of $[\![e]\!]$.



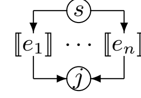
- *Task application*: functions $f :: \alpha_1 \rightarrow \alpha_2 \rightarrow \ldots \alpha_n \rightarrow \mathit{Task}\ \alpha$ $(n \geq 0)$ defined as $f\ a_1\ a_2 \ldots a_n\ =\ e$ applied to arguments $e_1 \ldots e_n$ are depicted as nodes in the workflow graph, shown on the right. Arguments $e_1 \ldots e_n$ can be any Clean expression. Higher order tasks are supported, the higher-order argument is then represented as a directed graph.



- *Sequential composition*: $[\![a \texttt{ >>| } b]\!]$ and $[\![a \texttt{ >>= } \lambda v \rightarrow b]\!]$ are represented as shown on the right. Variable v is in scope of $[\![b]\!]$.
- *Case distinction*: $[\![\textbf{case}\ e\ \textbf{of}\ p_1 \rightarrow e_1\,;\ \ldots\,;\ p_n \rightarrow e_n\,;]\!]$ $(n \geq 1)$. Here, e and e_i can be any Clean expression, and p_i can be any pattern. Each pattern variable introduced in p_i is in scope of $[\![e_i]\!]$.



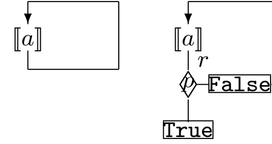
– *Parallel composition*: parallel composition of tasks is depicted by a *split connector* s, one or more branches $e_1 \dots e_n$ and a *join connector* j, as shown on the right. Pairs of split and join connectors are mapped to iTask combinators. Different types of join connectors are used for different types of joins, see the table below.



Split	Join	iTask combinator	Result type
$\wedge$	$\vee_{first}$	`anyTask` $[e_1, \dots, e_n]$	`Task` a
$\wedge$	$\vee_{left}$	e_1 `-||` e_2	`Task` a
$\wedge$	$\vee_{right}$	e_1 `||-` e_2	`Task` a
$\wedge$	$\wedge_{(,)}$	e_1 `-&&-` e_2	`Task` (a, b)
$\wedge$	$\wedge_{[]}$	`allTasks` $[e_1, \dots, e_n]$	`Task` $[a]$

The full paper will explain these combinators.

– *Iteration*: $[\![$`forever` $a]\!]$ and $[\![a$ `<!` $(\lambda r \rightarrow p)]\!]$ (repeat ... until) are represented as shown to the right. The variable r is in scope of p, but no further.



– *Multiple instances*: Many WDLs allow multiple instances of the same task to be started with different parameters. In iTask, this is accomplished by applying list combinators (such as `sequence` and `allTasks`) to lists of tasks created with different parameters. Gin defines tasks for these list combinators, and has a graphical representation for static lists expressions and list comprehensions.
Due to their length, these diagrams are omitted from this extended abstract, but will appear in the full paper.

Workflow models expressed in Gin are a hybrid form of graphical constructs and textual Clean expressions. Composition of iTask combinators is expressed graphically, while patterns and first-order task arguments are denoted as text. Ordinary Clean functions can be embedded in Gin models. These do not have a grapical representation.

3 The Gin tool

The Gin tool is a proof-of-concept implementation. In this section we discuss the following major parts of the tool: the front-end (Section 3.1), the implementation of the front-end (Section 3.2), the compilation of a constructed iTask workflow by Gin (Section 3.3), and the handling and reporting of errors (Section 3.4).

3.1 The front-end

The Gin front-end (Figure 1) is a web-based editor; it is implemented as a Java applet consisting mainly of a drawing canvas and a repository. A new workflow starts with a blank canvas. Nodes can be added to the workflow by dragging

them from the repository to the canvas. The editor supports common editing operations one would expect from a workflow editor like moving nodes, adding connectors etc.

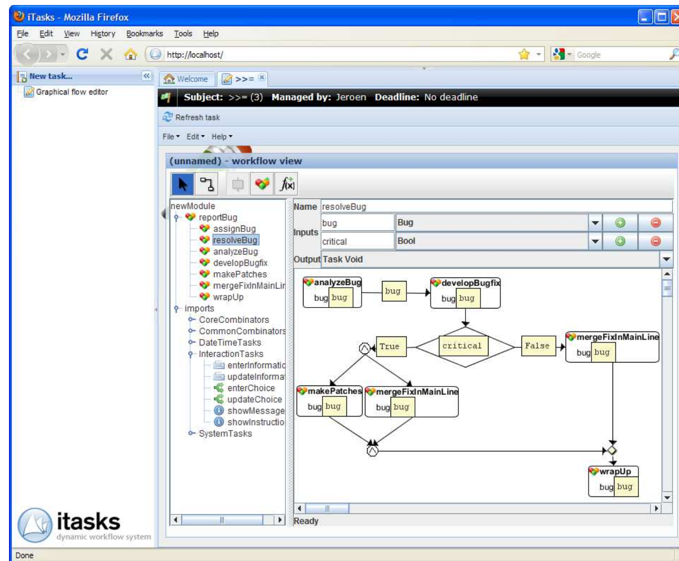


Fig. 1. Gin front-end showing a bug reporting workflow

3.2 Integration of Gin in iTask

Gin workflows graphs are stored as values of an algebraic data type named `GraphicalWorkflow`. These values contain all information to visually reconstruct their workflow diagrams, including layout.

In iTask, the function `updateInformation` can generate a default editor for any data type. This function creates an interactive element using a couple of generic functions: `gVisualize` for generating editor GUIs and `gUpdate` for updating edited values. With the use of specialization, we have created a specialized editor for `GraphicalWorkflow` values (as shown in Figure 1). Hence, editing Gin workflows boils down to editing values of type `GraphicalWorkflow`. The entire process around the manipulation of `GraphicalWorkflow` values can be regarded as a workflow itself - a meta-workflow. This meta-workflow is modeled in iTask; its main part consists of the task `updateInformation wf`, where `wf :: GraphicalWorkflow`.

3.3 Compiling Gin workflows to iTask workflows

The Gin tool compiles a `GrapicalWorkflow` value to an executable iTask in five steps. First, a block detection algorithm parses sequential blocks, parallel blocks

and structured loops in the graph and maps these to nodes in a block tree structure. Second, the block tree is transformed to an abstract syntax tree (AST) containing an iTask expression of type $f :: \alpha_1 \rightarrow \alpha_2 \rightarrow \ldots \alpha_n \rightarrow \mathit{Task}\ \alpha$. This transformation uses a set of bindings from graph node types to abstract syntax. Third, the AST is printed to concrete Clean source code, which is written to a file. The source file additionally contains a `Start` function which wraps the iTask expression in a **dynamic** [2] and writes it to a file. Fourth, the Gin tool calls the Clean compiler to compile the source file and runs the resulting executable, so the **dynamic** is written to disk. Finally, the **dynamic** file is loaded by the Gin tool, ready to be executed.

3.4 Error handling

During editing, the user gets immediate feedback about the incorrect parts of the model. This is implemented by means of a compiler feedback loop. After each editing operation, the iTask server starts the compilation process in the background. Compilation may fail for several reasons. Either parsing fails because the graph structure is not well formed, or the generated source code is erroneous, so the Clean compiler outputs error messages. The Gin tool keeps track of a map from the line numbers in the compiled source code back to graphical nodes. If errors are found, the incorrect graph nodes are highlighted. When the mouse cursor is moved over these nodes, a text box with the error message is shown.

4 Conclusions

The Gin language is a graphical notation for a subset of the iTask WDL. Gin makes iTask models more accessible for domain experts who may be unfamiliar with functional programming. Gin captures the iTask control flow and workflow decomposition in a grapical notation resemblant to conventional WDLs. However, task parameters and operations on data structures are still expressed as Clean expressions.

Our proof-of-concept implementation shows that the iTask system provides a suitable environment for embedding custom editors for complex data types like workflow graphs, including feedback for error handling.

In our experience, graphical front-ends like Gin make good use cases for accessing the compiler via an API. If an API had been available for the Clean compiler, we could pass generated ASTs directly to the compiler, instead of having to print the ASTs, write source files which are read and parsed again by the compiler. Besides, compiler errors can be passed in type-safe way.

References

1. W.M.P. van der Aalst. Chapter 10: Three good reasons for using a Petri-net-based Workflow Management System. volume 428 of *The Kluwer International Series in Engineering and Computer Science*, pages 161–182, Boston, Massachusetts, 1998. Kluwer Academic Publishers.

2. Marco Pil. Dynamic types and type dependent functions. In Kevin Hammond, Tony Davie, and Chris Clack, editors, *Proceedings of the 10th International Workshop on the Implementation of Functional Languages, IFL '98, London, UK*, volume 1595 of *Lecture Notes in Computer Science*, pages 169–185. Springer-Verlag, 1999.
3. Rinus Plasmeijer, Peter Achten, and Pieter Koopman. iTasks: executable specifications of interactive work flow systems for the web. In *Proceedings of the 12th International Conference on Functional Programming, ICFP '07*, pages 141–152, Freiburg, Germany, 1-3, October 2007. ACM Press.