Evolution of a Parallel Task Combinator

Bas Lijnse

Radboud University Nijmegen
b.lijnse@cs.ru.nl

Abstract. The development of experimental software is rarely straightforward. If you start making something you don't understand yet, it is very unlikely you get it right at the first try. The iTask system has followed this predictably unpredictable path. In this system, where combinator functions are used to construct interactive workflow support systems, the core set of combinator functions has changed along with progressed understanding of the domain. Continuous work on this system led to the emergence of a new programming paradigm for interactive systems: Task-Oriented Programming (TOP). In this paper we reconstruct the evolution of one of the core iTasks combinators to catch a glimpse of this emergence. The combinator is the `parallel` combinator that facilitates the composition of multiple concurrent tasks into a single one. We reconstruct its evolution from the written record in the form of published papers and discuss this reconstruction and what it tells about the progressed understanding of programming with tasks.

1 Introduction

If you don't know where you are going, you don't know where you will end up. Making research software based on ideas that you don't yet fully understand is inherently different from "production" software where you assume that you can clearly scope the requirements, and you can draw up designs based on understood principles. Although you cannot reliably work towards a defined product, it does not mean that you are just randomly doing something. By trying to embody little understood ideas in a software system, and trying to make things of which you don't know if they are even possible you learn what is possible and you get a better understanding of your initial ideas. You have, of course, ideas about what you consider important and expectations of what you might find, but you have to try to keep an open mind and be prepared to change course midway. In the process you may find that your hypotheses about properties of your system don't hold or you end up with something different than you imagined.

The iTask System (iTasks) is a system that followed this uncertain path and has been changing ever since it was first conceived [9]. This system started out as a modest experiment to express workflow in the functional language Clean [16], but that was just the beginning. In the past years it evolved and eventually became a general-purpose framework for making interactive web-based multi-user applications, supporting a new programming paradigm: Task-Oriented Programming (TOP). In this paradigm, multi user systems are expressed by composing

tasks. The evolution of iTasks was driven primarily by two desires. The first driver was the quest to find a minimal, yet complete, core set of primitives to express task patterns with. The second driver was the wish to not be limited to work with little variation, modeled by rigid workflows, but be able to capture a wide range of dynamic real-world tasks. These desires pushed the scope of the iTask system beyond what is usually considered workflow, and into a general purpose framework for any interactive system. At some point we realized that programming applications with iTasks had drifted so far away from workflow specification and traditional functional programming, that it could be considered a separate new paradigm. The present day iTask system provides a combinator based embedded domain specific language that implements the basic TOP constructs for defining and composing tasks. It consists of generic user interaction primitives, combinators for sequential and parallel composition of tasks and primitives for sharing data between tasks.

Some iTask combinators have remained relatively stable, such as the monadic "bind" for sequential compositions. Others however, changed more often. One of the combinators that changed a lot was the `parallel` core combinator for concurrent execution of multiple tasks. This combinator originally was not even a single combinator but a set of specialized combinators that were more or less related to each other. At some point it seemed that there was a new version of this combinator in every published paper. When students who used iTasks for their research projects asked for a paper they could read as introduction, we almost always had to answer that there was of course something they could read, but that the latest version was already different from that paper.

Now that we have arrived at the TOP paradigm, it is interesting to look back and see how we got here. We usually focus on the future, and aim to improve the status quo, but sometimes looking back can provide valuable insights too. A festschrift such as this provides a good opportunity to do so. In this paper we reconstruct the evolution of the `parallel` combinator as angle on the emergence of TOP. Because we cannot rely on memory, we turn to the accumulated publications about iTasks as written record of the evolution of the system, and to keep the scope manageable, we focus on the "parallel" combinator in particular. In the remainder of this paper, we see how we can reconstruct the history of the parallel combinator (Section 2), walk through its evolution (Section 3), and reflect on it (Section 4).

2 Methodology

To reconstruct the evolution of the `parallel` combinator, we have two sources at our disposal. There is a written record of milestone versions in the form of publications that contain an explanation of the iTask system. The advantage of these publications is that they do provide an explanation along with the definitions. The disadvantage of publications is that it is hard to reconstruct the time frame in which the published definition was used due to the delays of the publication process. Fortunately, since the majority of the publications were published in

conference proceedings, the submission deadlines of those conferences are mentioned in their calls for papers. These calls are easily retrievable through public archives of mailing lists to which they were posted. The submission deadlines provide a reasonable estimation of the date the paper was finalized.

The second source we can use is the logging provided by the public Subversion (version control) repository of the iTask system. This record is more fine grained than the publication record, because not all small changes are immediately worthy of publication. Finding out when changes were made is very easy for this source, because all commits to the repository are automatically timestamped. The disadvantage of this source is that changes are often only accompanied by short log messages, such that we need to study the source code of the system at critical points in time to understand the interface and semantics of the combinator in that time frame.

In this paper we limit ourselves to the publication record because these are milestone versions and we aim to reconstruct the complete history. The Subversion log is simply too fine grained for such a global overview. To make our reconstruction we simply collect all publications about the iTask System that mention the parallel combinator or its specialized predecessors. We then organize them chronologically and isolate those sections of the papers that explain parallel composition.

3 Evolution of the Parallel Combinator

In this section we present the history of the `parallel` as reconstructed from publicly available sources. Other than grouping the chronology in related periods, we do not yet interpret anything. We only collect and organize what is said about the combinators and postpone discussion to the next section.

3.1 The 'AND' and 'OR' Period

The `parallel` combinator did not start out as a single construct for all possible parallel patterns of combining tasks. It started out as a set of combinators for specific patterns. The first iTasks paper, "iTasks: Executable Specifications of Interactive Work Flow Systems for the Web" [9] presented at ICFP 2007 (October 1-3, 2007), defined two combinators for parallel composition. They are explained as follows:

> The infix operator (`t1 -&&- t2`) activates subtasks t1 and t2 and ends when both subtasks are completed; the infix operator (`t1 -||- t2`) also activates two subtasks t1 and t2 but ends as soon as one of them terminates, but it is biased to the first task at the same time. In both cases, the user can work on each subtask in any desired order. A subtask, like any other task, can consist of any composition of iTasks.

The paper also shows the implementations of these combinators. In this code we can already see that both combinators are very similar.

```
(-&&-) infixr 4 :: (Task a) (Task b) → Task (a,b) | iCreate a & iCreate b
(-&&-) taska taskb = doTask and
where and tst=:{tasknr}
      # (a,tst=:{activated=adone}) = mkParSubTask 0 tasknr taska tst
      # (b,tst=:{activated=bdone}) = mkParSubTask 1 tasknr taskb tst
      = ((a,b),set_activated (adone && bdone) tst

(-||-) infixr 3 :: (Task a) (Task a) → Task a | iCreate a
(-||-) taska taskb = doTask or
where or tst=:{tasknr}
      # (a,tst=:{activated=adone}) = mkParSubTask 0 tasknr taska tst
      # (b,tst=:{activated=bdone}) = mkParSubTask 1 tasknr taskb tst
      = ( if adone a (if bdone b createDefault)
        , set_activated (adone || bdone) tst
        )
mkParSubTask :: Int TaskID (Task a) → Task a
mkParSubTask i tasknr task = task o newSubTaskNr o set_activated True o subTaskNr i
```

In the lecture notes of the CEFP 2007 summer school [8], which included a course on iTasks, we find the exact same code and accompanying explanations as in the ICFP 2007 paper. This summer school was held the same summer, so this is not surprising.

The next publication that mentions the parallel iTask combinator is "Declarative Ajax and Client Side Evaluation of Workflows using iTasks" presented at PPDP 2008 (July 15,17 2008) [13]. This paper gives type definitions for the same two parallel combinators:

```
(-||-) infixr 3 :: (Task a) (Task a) → Task a | iData a
(-&&-) infixr 4 :: (Task a) (Task b) → Task (a,b) | iData a & iData b
```

The only difference between these signatures and the previous ones is the context of `iData` instead of `iCreate`. The difference between these restrictions is that the `iData` class also contains generic storage and visualization in addition to default value creation provided by `iCreate`. The paper provides explanations of the combinators by examples that reveal that the semantics of the combinators did not change:

> The expression `t-||-u` offers tasks t and u simultaneously. As soon as either one is finished first, `t-||-u` is also finished. Any work in the other task is discarded. The `-||-` combinator is very useful to express work that can be aborted by other workers or external circumstances. In
>
> `ot = yt -||- nt -||- et`
>
> the iTasks system offers the task `yt`, `nt`, and `et` simultaneously. Any edit work in `et` is discarded when the user presses one of the buttons labeled Yes or No.

And for the `-&&-` combinator:

> If one really needs both results of tasks `t` and `u`, then this is expressed by `t -&&- u`, which runs both tasks to completion and returns both results. For instance, if we need a string and an integer (with default value 5) we can use the task:

```
at :: Task (String, Int)
at = ot -&&- editTask "Done" 5
```

The next mention of the parallel combinators is in the lecture notes of the AFP 2008 summer school [10] (May 19-24, 2008). In this publication, the iTask system is explained following a case study. The combinators are initially only explained in so far they are relevant to that case. Therefore initially only the "OR" combinator is defined:

`(-||-)` **infixr** 3 `::` `(Task a)` `(Task a)` → `Task a | iData a`

And explained only briefly as:

> The left-biased task `t -||- u` is finished as soon as either `t` or `u` has finished, or both.

In a later section that explains the semantics of the iTask system using a simplified model, the 'AND' combinator is introduced together with its equivalent in the model:

> We introduce the iTask combinator `t -&&- u`, and represent it by `t .&&. u`. In the case study in Section 2 we did not use this combinator, but it belongs to the basic repertoire of the iTask system, therefore we include it here. In the task `t -&&- u`, both subtasks `t` and `u` are available to the user. The composite task is finished as soon as both subtasks are finished. Hence, it differs from `-||-` in which termination is controlled by the first subtask that finishes. Also, its type is more general, because the types of the return values of the subtasks are allowed to be different, the type of this operator in the iTask system is `(Task a)` `(Task b)` → `Task (a,b) | iData a & iData b`.

The full semantic model is too lengthy to quote here, but the reduction of the modeled combinators `.||.` and `.&&.` that represent the 'OR' and 'AND' combinators is defined to exhibit the behaviour that has been explained in the various papers so far.

This semantic model is worked out in full in the next publication that mentions the parallel combinators. In "An Executable and Testable Semantics for iTasks" [5] presented at IFL 2008 (September 10-12) we find the now familiar type signatures and the following explanation.

> The expression `t -||- u` indicates that both iTasks can be executed in any order and interleaved, the combined task is completed as soon as any subtask is done. The result is the result of the task that completes first, the other task is removed from the system. The expression `t -&&- u`

states that both iTasks must be done in any order (interleaved), the combined task is completed when both tasks are done. The result is a tuple containing the results of both tasks.

The most notable thing here is that the interleaving semantics of the combinators is mentioned explicitly for the first time.

3.2 Lists of Parallel Tasks

In the next publication, "Web Based Dynamic Workflow Systems and Applications in the Military Domain" [2], in the 2008 issue of NL ARMS, we see additional parallel combinators for the first time. The 'OR' and 'AND' combinators are generalized to versions that use lists of tasks that are executed in parallel. The combinators are explained by example:

> The AND (`-&&-`) operator generates two tasks that both have to be finished before the results can be used.

```
simpleAndMU :: Task Int
simpleAndMU
=       (0   @:: editTask "Number entered" 0)
  -&&- (1   @:: editTask "Number entered" 0)
  >>= λ(v,w) → 2 @:: editTask "Sum" (v+w)
```

> For AND also a multi-version '`andTasks`' exists, which handles a list of tasks. The task completes when all subtasks are completed.
> The OR (`-||-`) operator generates two tasks in parallel. As soon as one of them finishes the result of that task is available. The result of the other task is ignored.

```
simpleOrMU :: Task Int
simpleOrMU
=       (0   @:: editTask "A number" 0)
  -||- (1   @:: editTask "A number" 0)
  >>= λv → 2 @:: editTask "First number" v
```

> Also for OR a multi-version '`orTasks`' exists, which handles a list of tasks. The task completes as soon as one of the tasks completes.

Additionally, an important new concept is reported for the first time: a general parallel combinator with which other combinators can be expressed:

> In iTasks a special version of '`andTasks`' exists: '`andTasksCond`'. A number of tasks can be started in parallel. Each time one of the tasks is finished a condition is applied to all completed tasks. If the condition is met, '`andTasksCond`' is finished and the completed results are returned in a list.

This combinator is also explained by example:

```
simpleAndTaskCond :: Task Int
simpleAndTaskCond
=  andTasksCond pred [("User" +++ toString u,
                       u @:: editTask "Number entered" 0) \\ u ← [1..4]]
 ⋙ λxs → [Txt "Their sum is"] !>> return_D (sum xs)
where pred xs = sum xs > 3
```

> Here a parallel task for 4 users is started. They all have to enter a number. Here the condition checks if the sum of the already entered numbers is greater than 3. As soon as this is the case this task stops and the results are passed to another task where they are displayed.

What is more, the paper shows how this '`andTasksCond`' can be used to express other combinators:

> This is a very powerful combinator because many other combinators can be expressed using it. For example the definitions of '`andTasks`' and '`orTasks`' can be given by:

```
andTasks xs = andTasksCond (λys = length ys == length xs) xs
orTasks  xs = andTasksCond (λys = length ys == 1) xs
```

The next publication that mentions the parallel combinators is "Tasks 2: iTasks for End-users" [6] presented at IFL 2009 (September 23-25, 2009). Although it reports on a new implementation of the iTask system, the combinator language has not changed as can be seen in the signatures that are mentioned without further explanation.

```
// Execute two tasks in parallel
(-&&-) infixr 4 :: (Task a) (Task b) → Task (a,b)
// Execute two tasks in parallel, finish as soon as one yields a result
(-||-) infixr 3 :: (Task a) (Task a) → Task a
// Execute all tasks in parallel
allTasks :: ([Task a] → Task [a])
// Execute all tasks in parallel, finish as soon as one yields a result
anyTask :: ([Task a] → Task a)
```

The new combinators introduced here, '`allTasks`' and '`anyTask`' appear to be just variations of the '`andTasks`' and '`orTasks`' combinators in the NL ARMS paper.

This apparent variation is confirmed in the next publication, "Embedding a Web-Based Workflow Management System in a Functional Language" [4] presented at LDTA 2010 (March 27-28, 2010). The signatures with an explanation of these combinators are given.

```
// Splitting-joining any number of arbitrary tasks:
anyTask  :: [Task a] → Task a | iTask a
allTasks :: [Task a] → Task [a] | iTask a
```

> Any number of tasks `ts` = $[t_1...t_n](n >= 0)$ can be performed in parallel and synchronized (also known as splitting and joining of workflow expressions): `anyTasks ts` and `allTasks ts` both perform all tasks ts simultaneously, but `anyTasks` terminates as soon as one task of ts terminates and yields its value, whereas `allTasks` waits for completion of all tasks and returns their values.

In this paper the fully generalized `parallel` combinator is presented for the first time. Unlike the '`andTasksCond` combinator, which could not express '`anyTask`' for example because its type is always `Task [a]`, this combinator is capable of expressing all parallel patterns.

> As a final example, iTask provides a core combinator function, parallel that is used in the system to define many other split-join combinators such as `anyTask` and `allTasks` that were shown earlier. Its type signature is:
>
> ```
> parallel :: ([a] → Bool) ([a] → b) ([a] → b) [Task a] → Task b
>                                                 |iTask a & iTask b
> ```
>
> `parallel c f g ts` performs all tasks within `ts` simultaneously and collects their results. However, as soon as the predicate `c` holds for any current collection of results, then the evaluation of `parallel` is terminated, and the result is determined by applying `f` to the current list of results. If this never occurs, but all tasks within `ts` have terminated, then `parallel` terminates also, and its result is determined by applying `g` to the list of results.

The paper after this one reinforces the idea of a single general parallel combinator to express multiple patterns without going into details. This paper, "Web Based Dynamic Workflow Systems for C2 of Military Operations" [3], presented at ICCRTS 2010 (June 22-24, 2010) stresses the use of a single general concept and gives '`anyTask`' and '`allTasks`' as examples.

> An important combinator for executing a number of tasks in parallel is the `parallel` combinator. Where other workflow formalisms contain a large number of patterns for executing tasks in parallel, iTask needs only one combinator for this. Using the power of the functional host language, one can construct all other patterns (and more) using this single combinator. This is hard to do in other workflow languages because these lack the right abstraction mechanism for realizing this. With the parallel combinator one can start the execution of several tasks in parallel and stop this execution as soon as a user specified condition is fulfilled. For example, one can stop when one task (or-parallelism) is finished:
>
> `anyTask [task_1,task_2,task_3,task_n]`
>
> When all tasks (and-parallelism) are finished:
>
> `allTasks [task_1,task_2,task_3,task_n]`

Or when the results of the finished tasks satisfy a certain condition (ad-hoc parallelism):

```
conditionTasks condition [task_1,task_2,task_3,task_n]
```

These different combinators are all shorthands for the same generic `parallel` combinator instantiated with different parameters.

The next paper, "iTask as a New Paradigm for Building GUI Applications" [7] presented at IFL2010 (September 1-3, 2010), is concerned mostly with the additional concepts needed to make GUI programs with iTasks. It explains the iTasks combinators only to the extent necessary for the leading example of the paper. Regarding parallel combinators this is just the familiar 'AND' combinator.

Finally, we need a combinator to compose tasks in parallel: `-&&-` performs both tasks and returns their combined result when both are terminated.

```
(-&&-) infixr 4 :: (Task a) (Task b) → Task (a,b) | iTask a & iTask b
```

Similarly, the next paper also explains the combinators only to the extent necessary for the purpose of the paper. This paper, "iTasks for a Change: Type-Safe Run-Time Change in Dynamically Evolving Workflows" [11] presented at PEPM2011 (January 24-25, 2011), ignores that the 'AND' and 'OR' combinators are expressed using a general '`parallel`' combinator. It defines their semantics directly such that their behaviour during run-time change can be explained.

```
(-||-) infixr 3 :: (Task a) (Task a) → Taska | iTaska
(-&&-) infixr 4 :: (Task a) (Task b) → Task (a, b) | iTask a & iTask b
```

To compose tasks in parallel, the combinators `-||-` and `-&&-` are provided. A task constructed using `-||-` is finished as soon as either one of its subtasks is finished, returning the result of that task. The combinator `-&&-` is finished as soon as both subtasks are finished, and pairs their results.

The semantics of these combinators is defined in a separate semantic domain. The paper gives definitions for `-||-` as well as `-&&-`, but we will limit ourselves to the definition of `-||-`.

The semantic function of `-||-` is defined as follows:

```
(-||-) infixr 3 :: (STaska) (STaska) → STaska
(-||-) ta ua = λ i p e s →
 case ta (subIds i !! 0) p e s of
  (NF va, s) → (NF va, s)
  (Redex nta, s) →
    case ua (subIds i !! 1) p e s of
    (NFwa, s) → (NFwa, s)
    (Redex nua, s) → (Redex (nta -||- nua), s)
```

From the formal definition of the behaviour of the 'AND' and 'OR' combinators in this paper we can see that the semantics of these combinators have not changed since the original definitions three and a half year earlier.

3.3 Towards Dynamically Extensible Parallel Tasks

The next paper covers the overall design of the iTask system again. This paper, "Getting a Grip on Tasks that Coordinate Tasks" [14] was an invited paper at the LDTA 2011 Workshop (March 26-27, 2011). It both explains the status quo of the iTask system and discusses future needs. This paper starts by giving the familiar definitions for the basic 'AND' and 'OR' combinators. When discussing the expressiveness of the combinator language, the general `parallel` combinator is explained and definitions for 'AND' and 'OR' are given.

> The need for more functionality does not necessarily imply that more combinators are required. By using higher order functions, Swiss-Army-Knife combinators can be defined, that strongly reduce the number of needed core combinators. In the current iTask system, the parallel combinator is one such example:
>
> ```
> parallel :: ([a] → Bool) ([a]→b) ([a]→b) [Task a] → Task b
>                                                   | iTask a & iTask b
> ```
>
> For instance, the core combinators `-||-` and `-&&-` can be replaced by suitable parametrization of `parallel`. The function `parallel predOK someDone allDone taskList` takes a list of tasks (`taskList`) to be executed in parallel, a predicate (`predOK`), and two conversion functions (`someDone` and `allDone`). Whenever a member of `taskList` is finished, its result is collected in a list `results` of type `[a]`, maintaining the order of tasks. Now `predOK results` is computed to determine whether `parallel` should complete, in which case the result is computed by `someDone results`. When all parallel tasks have run to completion, and `predOK` is still not satisfied, then `parallel` also completes, but now with result `allDone results`. We can define `-||-` and `-&&-` as follows:
>
> ```
> (-||-) infixr 3 :: (Task a) (Task a) → Task a | iTask a
> (-||-) ta1 ta2 = parallel (not o isEmpty) first undef [ta1, ta2]
> where
>  first [a] = a
>
> (-&&-) infixr 4 :: (Task a) (Task b) → Task (a, b) | iTask a & iTask b
> (-&&-) ta tb = parallel (const False) undef all
>                         [ta >>= Left, tb >>= Right]
> where
>   all [Left a,Right b] = (a,b)
> ```
>
> Although a Swiss-Army-Knife combinator such as `parallel` can be used to define many different kinds of parallel behaviours, there is room for improvement here as well. With `predOK` one can freely define when the parallel tasks can be stopped, but perhaps one also needs to be able to start new tasks dynamically, because more work is required.

So far the paper provides little new information. However in the section that addresses future needs, it is revealed that big changes to the `parallel` combinator are afoot.

The workflow engineer should be able to specify the means of control as (arbitrarily many) additional tasks that coordinate these tasks. We hypothesize that these forms of parallel behaviour can be captured with a single, more general combinator. The combinator needs to meet the following criteria:

1. The number of tasks in the current `parallel` combinator remains constant, and `parallel` can only enforce early termination, not the extension of new tasks. The number of tasks in a parallel setting should not be fixed once and for all, but should adapt to the needs of the current situation.
2. The tasks within the current parallel combinator simply perform their duty and as such do not interfere with each other (except of course when using shared communication). Next to these regular tasks we introduce control tasks. These are also tasks, but, being control tasks, they edit the collection of parallel tasks. In this way, we can replace the predefined behaviour of task delegation and instead leave it to the workflow engineer whether or not to use a predefined control delegation-task or introduce a (number of) custom control task(s).
3. Because the number of both regular and control tasks varies during the evaluation of a parallel group, we need to share information about the state of the parallel group. Access to this state is restricted to control tasks only, which is easily achieved using the strong type system.
4. In the current `parallel` combinator, control is limited to either early completion (computed by `predOK`) in which case the final task result was computed by `someDone` or full completion in which case the final result was computed by `allDone`. In the more general case, we need to decide how to continue whenever a regular or control task runs to completion. Again, this should not be computed by the regular tasks. Instead, we need a function that knows which task has completed, and hence has a result value that needs to be accumulated in the shared state. In addition, this function can decide what should happen with the group of parallel (control and regular) tasks: tasks can be suspended and resumed, they can be removed, replaced, and new (control and regular) tasks can be added to the group of parallel tasks. It is clear that this functionality subsumes the current behaviour of parallel, and adds behaviour that was inexpressible before.
5. The final part that should be abstracted from is the arrangement, or layout, of the generated GUIs of the (control and regular) tasks. In the current iTask system a distinction is made between a parallel form for tasks that can, in principle, each be delegated to other workers and a parallel form for tasks which GUI should be merged into one single presentation. In order to abstract from this, it is better to parameterize the new parallel combinator with a function

that describes how the component GUIs of (control and regular) tasks should merged.

We are currently experimenting with a single `parallel` combinator that meets the above criteria. With this combinator we hope to express all other task combinators as special cases.

These five points illustrate that all aspects of parallel combination are considered. Even the visual representation (layout) of parallel combinations, which we have not encountered in the publications so far, is taken into account.

In the next publication, "Defining Multi-user Web Applications with iTasks" [12] in the lecture notes of the CEFP 2011 summer school (June 14-24, 2011), we can see that the proposed changes to the parallel combinator have found their way into the system. In this paper two sections (8 & 9) are devoted to the parallel combinator. The first of those presents a far more complex `parallel` combinator:

> The iTask system provides a single, swiss army knife combinator for this purpose, called parallel. In this section we explain how to use this versatile combinator for an arbitrary, yet constant, number of users. In Section 9 we continue our discussion and show how it can be used to accommodate a dynamic number of users. The signature of parallel is:
>
> ```
> parallel :: d s (ResultFun s a) [TaskContainer s] → Task a
>                                                   | iTask s & iTask a
>                                                   & descr d
> ```
>
> We briefly discuss its parameters first. The first parameter is the usual description argument that we have encountered many times so far. It plays the same role here: a description to the user to inform her about the purpose of this particular parallel task in the workflow. The second argument is the initial value of the state of the parallel task: the state is a shared data that can be inspected and altered only by the tasks that belong to this parallel task. The third argument is a function of type:
>
> ```
> :: ResultFun s a :== TerminationStatus s → a
> :: TerminationStatus = AllRunToCompletion | Stopped
> ```
>
> The purpose of the `ResultFun` function is to turn the value of the state of the parallel task at termination into the final value of the parallel task itself. They need not have the same type, so the state is converted to the final value when the parallel task is finished. The `parallel` combinator can terminate in two different ways. It can be the case that all subtasks are finished (`AllRunToCompletion`). But, as we will see later, a subtask can also explicitly kill the whole parallel construction (`Stopped`). This information can be used to create a proper final value of parallel. Finally, the fourth argument is the initial list of task (container)s that constitute the parallel task. A task container consists of two parts: a task type representation (`ParallelTaskType`) defining how the subtask relates to its super-task, and the subtask itself (defined on shared state `s`) to be run in parallel with the others (`ParallelTask s`):

```
:: TaskContainer s :== (ParallelTaskType, ParallelTask s)
:: ParallelTaskType = Embedded
                    | Detached ManagementMeta
```

The `ParallelTaskType` is either one of the following:

- `Embedded` basically 'inlines' the task in the current task.
- `Detached meta` displays the task computed by the function as a distinct new task for the user identified in the worker field of meta. `ManagementMeta` is a straightforward record type that enumerates the required information:

```
:: ManagementMeta =
    { worker         :: Maybe User
    , role           :: Maybe Role
    , startAt        :: Maybe DateTime
    , completeBefore :: Maybe DateTime
    , notifyAt       :: Maybe DateTime
    , priority       :: Maybe TaskPriority
    }
:: TaskPriority = HighPriority | NormalPriority | LowPriority
```

It should be noted that the `u @:` combinator is simply expressed as a parallel combination of two tasks. One of type `Detached` with the worker set, and another of type `Embedded` that displays progress information.

```
:: ParallelTask s :== (TaskList s) → Task ParallelControl
:: TaskList s
:: ParallelControl = Stop | Continue
```

The task creation function takes as argument an abstract type, `TaskList s`, where `s` is the type of the data the subtasks share. Every subtask has to yield a task of type `ParallelControl` to tell the system, when the subtask is finished, whether the parallel task as a whole is also finished (by yielding `Stop`) or not (by yielding `Continue`.) As will be explained in Section 9, the number of subtasks in the task list can change dynamically. One can enquire its status, using the following functions on the abstract type `TaskList s`:

```
taskListState      :: (TaskList s) → Shared s | TC s
taskListProperties :: (TaskList s) → Shared [ParallelTaskInfo]
```

With the function `taskListState` one can retrieve the data shared between the tasks of the `parallel` combinator. As discussed in Section 5, you can use `get`, `set`, and `update` to access its value. There is another function, `taskListProperties`, which can be used to retrieve detailed information about the current status of the parallel tasks created. This can be used to control the tasks, and is explained in more detail in the next section.

The second section devoted to the `parallel` combinator in this paper covers dynamically adding and removing tasks from a parallel set:

In this section it is shown how the `taskList` can be used to dynamically alter the number of subtasks running in parallel. The following operations are offered to the programmer.

```
appendTask :: (TaskContainer s) (TaskList s) → Task Int | TC s
removeTask :: Int (TaskList s) → Task Void → TC s
```

Tasks can be appended to the list of tasks running under this parallel construction using `appendTask`. In a similar way, `removeTask` terminates the indicated task from the list of tasks, even if it has not run to completion.

The publication after this large change concerns itself with the more friendly derived parallel combinators only. In this paper, "GiN: A Graphical Language and Tool for Defining iTask Workflows" [1] presented at TFP 2011 (May 16-18, 2011), only the following familiar signatures and explanation are presented:

```
(-||-) infixr 3 :: (Task a) (Task a) → Task a      | iTask a
(-&&-) infixr 4 :: (Task a) (Task b) → Task (a,b) | iTask a & iTask b
anyTask         :: [Task a]          → Task a      | iTask a
allTasks        :: [Task a]          → Task [a]    | iTask a
```

Tasks can be composed in parallel. Either the result of the first completed task is returned (`-||-` and `anyTask` combinators) or the results of all parallel tasks are collected and returned as a whole (`-&&-` and `allTasks`)

In the final and most recent publication that covers the `parallel` combinator we see a new definition again. In this publication, "Task-Oriented Programming in a Pure Functional Language" [15] presented at PPDP2012 (May 31, 2012), the `parallel` combinator is presented as follows:

Tasks can often be divided into parallel sub tasks if there is no specific predetermined order in which the sub tasks have to be done. It might not even be required that all sub tasks contribute sensibly to a stable result. All variants of parallel composition can be handled by a single parallel combinator:

```
parallel:: d → [(ParallelTaskType,ParallelTask a)]
           → Task [ (TimeStamp, Value a) ] | descr d & iTask a

:: ParallelTaskType = Embedded | Detached ManagementMeta
:: ManagementMeta = { worker :: Maybe User
                    , role   :: Maybe Role
                    , ...
                    }
:: ParallelTask   a== SharedTaskLista → Taska
:: SharedTaskList a== ROShared (TaskList a)
:: TaskList a = { state :: [Value a]
                , ...
                }
```

> We distinguish two sorts of parallel sub-tasks: `Detached` tasks get distributed to different users and `Embedded` tasks are executed by the current user. The client may present these tasks in different ways. `Detached` tasks need a window of their own while embedded tasks may by visualized in an existing window. With the `ManagementMeta` structure properties can be set such as which worker must perform the sub-task, or which role he should have. Whatever its sort, every parallel sub-task can inspect each others progress. Of each parallel sub-tasks its current task value and some other system information is collected in a shared task list. The parallel sub-tasks have read-only access to this task list. The `parallel` combinator also delivers all task values in a list of type `[(TimeStamp,Value a)]`. Hence, the progress of every parallel sub-task can also be monitored constantly from the "outside".

The paper does not go into details of adding and removing tasks but mentions that it is still possible.

> For completeness, we remark that the shared task list is also used to allow dynamic creation and deletion of parallel sub-tasks. We do not discuss this further in this paper.

3.4 Summary

Because the chronology given in this section may be too much to take in at once, Table 1 summarizes the results presented in this section. It dates the publications which are identified by the conference or journal acronym, it indicates which parallel combinators are covered in that paper, and some of the properties the parallel combinator(s) had according to that paper. These properties are: The use of parameters to make derived combinators easier, whether the `parallel` has a variable number of tasks, and if there is data sharing between branches in a parallel combination.

4 Reflections

Now that we have reconstructed the history of the parallel combinator, we can discuss the developments it went through in the five years worth of publications in which it is mentioned.

4.1 Towards a Unified Parallel

The parallel combinator in the last publication we examined [15] is something completely different than the simple orginal 'AND' and 'OR' [9]. Yet as we read through the history, the changes are mostly gradual. The semantics of the 'AND' and 'OR' remain unchanged for a long time, but with the introduction of the `andTasksCond` combinator [2] and later the `parallel` combinator [4], a single unified combinator emerges that aims to capture *all* parallel constructs. Once this unified combinator is established it is clear that there can be a single core combinator for all possible task combinations.

Paper	Date (Deadline date)	AND,OR	ANY,ALL	parallel	Parameters	Var # tasks	Data sharing
The 'AND' and 'OR' Period							
ICFP2007 [9]	October 1-3, 2007 (April 6, 2007)	X	-	-	-	-	-
CEFP2007 [8]	June 23-30, 2007	X	-	-	-	-	-
PPDP2008 [13]	July 15-17, 2008 (April 10, 2008)	X	-	-	-	-	-
AFP2008 [10]	May 19-24, 2008	X	-	-	-	-	-
IFL2008 [5]	September 10-12 2008 (November 14, 2008)	X	-	-	-	-	-
Lists of Parallel Tasks							
NLARMS2008 [2]	September 2008	X	X	-	X	-	-
IFL2009 [6]	September 23-25, 2009 (November 1, 2009)	X	X	-	X	-	-
LDTA2010 [4]	March 27-28, 2010 (December 4, 2009)	-	X	X	X	-	-
ICCRTS2010 [3]	June 22-24 2010 (April 21, 2010)	-	X	X	X	-	-
IFL2010 [7]	September 1-3, 2010 (October 25, 2010)	X	-	-	-	-	-
PEPM2011 [11]	January 24-25, 2011 (October 22, 2010)	X	-	-	-	-	-
Towards Dynamic Extensible Parallel Tasks							
LDTA2011 [14]	March 26-27, 2011 (December 22, 2010)	X	-	X	X	-	-
CEFP2011 [12]	June 14-24, 2011	-	-	X	X	X	X
TFP2011 [1]	May 16-18, 2011 (June 24, 2011)	X	X	-	-	-	-
PPDP2012 [15]	September 19-21, 2012 (May 31, 2012)	-	-	X	-	X	X

Table 1. Parallel definitions in publications

4.2 Safe Experiments

The convergence of the parallel combinators to a single core combinator that can be used to express specific parallel patterns, does not mean we no longer see the 'AND' and 'OR' combinators. An interesting observation is that the publications can be divided in two categories. Papers that report on overall progress of the system ([9, 8, 10, 2, 6, 4, 3, 14, 12, 15]), and papers that focus on a single experimental extension or a specific issue ([13, 5, 7, 11, 1]). In the second category of papers we often see the 'AND' and 'OR' combinators still being used. These easier to explain combinators are used to provide context of the iTask system, in favor of the more accurate but more complex generalized combinator.

4.3 The Paradigm Shift

After the introduction of the single `parallel` we can see a new tension building fueled by the drive to capture real-world dynamic tasks. The original iTask system was based on tasks that always completed. Only when a task was completed its result was available for further computation of the workflow. For parallel sets of tasks, this meant that they always had to terminate as full set in order to deliver a result. In [14] we see the first signs of dissatisfaction with this model when the need for more dynamic parallel sets is discussed. Tasks in a parallel composition

should be able to be monitored, and if necessary extended if additional work is needed. In [12] we see the first attempt to realize these goals, but the notion of terminating tasks is still maintained. This leads to a powerful, yet complicated swiss-army-knife combinator that can express more parallel constructs, but is quite difficult to use. Only when in [12] the TOP paradigm had fully emerged, the `parallel` combinator was simplified again. By then it was clear that treating tasks as units of work that have to be completed before you can use their results, was making compositions more difficult than necessary. Defining tasks as units of work that continuously produce (temporary) results that can be observed made it possible to fully reduce the `parallel` combinator to its essence: just executing a set of tasks in parallel.

4.4 The Future?

By following the path of the `parallel` combinator we have seen the emergence of the TOP paradigm as an incremental interaction between the ideas about programming with tasks, and their concrete embodyment in the implementation of the iTask system. A process that eventually led to a new definition of the notion of tasks to provide the basis for a new way of programming interactive systems. With a major change in the `parallel` definition in the last publication we examined, it is too soon to tell whether this is the final one. For now at least it looks like the pieces of the puzzle have fallen into place and we have found a simple, yet powerful unified parallel construct.

References

1. J. Henrix, R. Plasmeijer, and P. Achten. GiN: a graphical language and tool for defining iTask workflows. In R. Peña, editor, *Proceedings of the 12th Symposium on Trends in Functional Programming, TFP '11, Selected Papers*, volume 7193 of *LNCS*, Madrid, Spain, 2012. Springer.
2. J. Jansen, P. Koopman, and R. Plasmeijer. Web based dynamic workflow systems and applications in the military domain. In T. Hupkens and H. Monsuur, editors, *Netherlands Annual Review of Military Studies - Sensors, Weapons, C4I and Operations Research*, pages 43–59, 2008.
3. J. Jansen, B. Lijnse, R. Plasmeijer, and T. Grant. Web based dynamic workflow systems for C2 of military operations. In *Revised Selected Papers of the 15th International Command and Control Research and Technology Symposium, ICCRTS '10*, Santa Monica, CA, USA, June 2010.
4. J. Jansen, R. Plasmeijer, P. Koopman, and P. Achten. Embedding a web-based workflow management system in a functional language. In C. Brabrand and P. Moreau, editors, *Proceedings 10th Workshop on Language Descriptions, Tools and Applications, LDTA '10*, pages 79–93, Paphos, Cyprus, March 27-28 2010.
5. P. Koopman, R. Plasmeijer, and P. Achten. An executable and testable semantics for iTasks. In S.-B. Scholz, editor, *Proceedings of the International Symposium on the Implementation and Application of Functional Languages, IFL '08, Hertfordshire, UK*, pages 53–64. University of Hertfordshire, 2008.

6. B. Lijnse and R. Plasmeijer. iTasks 2: iTasks for End-users. In M. Morazán and S. Scholz, editors, *Revised Selected Papers of the International Symposium on the Implementation and Application of Functional Languages, IFL '09, South Orange, NJ, USA*, volume 6041 of *LNCS*, pages 36–54. Springer-Verlag, 2010.
7. S. Michels, R. Plasmeijer, and P. Achten. iTask as a new paradigm for building GUI applications. In J. Hage and M. Morazán, editors, *Proceedings of the 22nd International Symposium on the Implementation and Application of Functional Languages, IFL '10, Selected Papers*, volume 6647 of *LNCS*, pages 153–168, Alphen aan den Rijn, The Netherlands, 2011. Springer.
8. R. Plasmeijer, P. Achten, and P. Koopman. An introduction to iTasks: defining interactive work flows for the web. In *Proceedings of the 2nd Central European Functional Programming School, CEFP '07*, Cluj-Napoca, Romania, 23-30, June 2007.
9. R. Plasmeijer, P. Achten, and P. Koopman. iTasks: executable specifications of interactive work flow systems for the web. In R. Hinze and N. Ramsey, editors, *Proceedings of the International Conference on Functional Programming, ICFP '07*, pages 141–152, Freiburg, Germany, 2007. ACM Press.
10. R. Plasmeijer, P. Achten, P. Koopman, B. Lijnse, and T. van Noort. An iTask case study: a conference management system. In P. Koopman, R. Plasmeijer, and D. Swierstra, editors, *Revised Lectures of the International Summer School on Advanced Functional Programming, AFP '08, Heijen, The Netherlands*, volume 5832 of *LNCS*, pages 306–329. Springer-Verlag, 2008.
11. R. Plasmeijer, P. Achten, P. Koopman, B. Lijnse, T. van Noort, and J. van Groningen. iTasks for a change - Type-safe run-time change in dynamically evolving workflows. In S. Khoo and J. Siek, editors, *Proceedings of the Workshop on Partial Evaluation and Program Manipulation, PEPM '11, Austin, TX, USA*, pages 151–160. ACM Press, 2011.
12. R. Plasmeijer, P. Achten, B. Lijnse, and S. Michels. Defining multi-user web applications with iTasks. In V. Zsók, Z. Horváth, and R. Plasmeijer, editors, *Proceedings of the 4th Central European Functional Programming School, CEFP '11, Revised Selected Papers*, volume 7241 of *LNCS*, pages 46–92, Eötvös Loránd University, Budapest, Hungary, 14-24, June 2012. Springer.
13. R. Plasmeijer, J. Jansen, P. Koopman, and P. Achten. Declarative Ajax and client side evaluation of workflows using iTasks. In *Proceedings of the 10th International Conference on Principles and Practice of Declarative Programming, PPDP '08*, pages 56–66, Valencia, Spain, 15-17, July 2008.
14. R. Plasmeijer, B. Lijnse, P. Achten, and S. Michels. Getting a grip on tasks that coordinate tasks. In *Proceedings Workshop on Language Descriptions, Tools, and Applications (LDTA)*, Saarbrücken, Germany, March 26-27 2011.
15. 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.
16. R. Plasmeijer and M. van Eekelen. Clean language report (version 2.1). http://clean.cs.ru.nl, 2002.