Orchestrating the Internet of Things with Task-Oriented Programming

a purely functional rhapsody

Radboud Dissertation Series

**RADBOUD
UNIVERSITY
PRESS**

Orchestrating the Internet of Things with Task-Oriented Programming

a purely functional rhapsody

Promotor:

    prof. dr. ir. M.J. (Rinus) Plasmeijer

Copromotoren:

    dr. J.M. (Jan Martin) Jansen (Nederlandse Defensie Academie)
    dr. P.W.M. (Pieter) Koopman

Manuscriptcommissie:

    prof. dr. S.-B. (Sven-Bodo) Scholz
    prof. dr. G.K. (Gabriele) Keller (Universiteit Utrecht)
    prof. dr. M. (Mary) Sheeran (Chalmers Tekniska Högskola, Zweden)

The world is indeed comic, but the joke is on mankind.

———

H.P. (Howard) Lovecraft

Рукописи не горят.
(Manuscripts don't burn.)

———

М.А. (Михаил) Булгаков
(M.A. (Mikhail) Bulgakov)

You start a question, and it's like starting a stone. You sit quietly on the top of a hill; and away the stone goes, starting others; ...

———

R.L. (Robert) Stevenson

I never thought, when I used to read books, what work it was to write them... It's work enough to read them sometimes... As to the writing, it has its own charms.

———

C.J.H. (Charles) Dickens

als ik dan van al dat schrijven
toch wel erg ben afgemat
ga ik lekker zitten lezen
o, wat heerlijk lijkt mij dat

———

H.M. (Dick) Bruna

Contents

voor Roos, Lotte en Elvira

Chapter 1

Prelude

This chapter introduces the dissertation by providing:

- a general introduction to the topics and research directions;
- a reading guide;
- background material on the internet of things, domain-specific languages, task-oriented programming, iTask, and mTask;
- and a detailed overview of the scientific contributions of this dissertation.

This dissertation is about orchestrating internet of things (IoT) systems safely and efficiently. There are at least 13.4 billion devices connected to the internet at the time of writing (Transforma Insights, 2023). Each of these devices sense, act, or otherwise, interact with people, computers, and the environment. Despite their immense diversity, they are all computers and they all require software to operate.

An increasing number of these connected devices are so-called edge devices that operate in the IoT. Edge devices are the leaves of the IoT systems. They perform the interaction with the physical world. It is not uncommon for edge devices to be physically embedded in the fabric itself. Typically, they reside in hard-to-reach places such as light bulbs, clothing, smart electricity meters, buildings, or even farm animals. The majority of edge devices are powered by microcontrollers. Microcontrollers are equipped with a lot of connectivity for integrating peripherals such as sensors and actuators. The connectivity makes them very suitable to interact with their surroundings. These miniature computers contain integrated circuits that accommodate a microprocessor designed for use in embedded applications. As a consequence, microcontrollers are cheap, tiny, have little memory, and contain a slow, but energy-efficient processor.

When coordinating an orchestra of edge devices, there is room for little error. Edge devices come and go, perform their own pieces, or are sometimes instructed to perform a certain piece, they might even operate without a central authority.

In a traditional setting, an IoT engineer has to program each device and their interoperation using different programming paradigms, programming languages, and abstraction levels. This results in semantic friction, which makes programming and maintaining IoT systems a complex and error-prone process.

This dissertation describes the research carried out around orchestrating these complex IoT systems using task-oriented programming (TOP). TOP is an innovative tierless programming paradigm for interactive multi-layered systems. By utilising advanced compiler technologies, much of the internals, communication, and interoperation between the tiers or layers of the applications are automatically generated. The compiler generates an application controlling all interconnected components from a single declarative specification of the required work. For example, the TOP system iTask is used to program all layers of multi-user distributed web applications from a single source specification. It is implemented in the general-purpose lazy functional programming language Clean, and therefore requires relatively powerful hardware. The inflated hardware requirements are no problem for regular computers but impractical for the average edge device.

This is where an additional domain-specific languages (DSLs) must play its part. DSLs are programming languages tailored to a specific domain. Consequently, jargon is not expressed in terms of the language itself, but is built into the language. Furthermore, the DSL can eschew language or system features that are irrelevant for the domain. Using DSLs, hardware requirements can be drastically lowered, even while maintaining a high abstraction level for the specified domain.

To incorporate the plethora of edge devices in the orchestra of a TOP system, the mTask system is used. The mTask language is a novel programming language for programming IoT edge devices using TOP. Where iTask abstracts away from the gritty details of multi-tier web applications, mTask has domain-specific abstractions for IoT edge devices, maintaining the high abstraction level that TOP offers. As it is integrated with iTask, it allows for all layers of an IoT application to be programmed from a single source.

1.1 Reading guide

This work is structured as a purely functional rhapsody. The Wikipedia contributors (2022) define a musical rhapsody as follows:

> A rhapsody *in music is a one-movement work that is episodic yet integrated, free-flowing in structure, featuring a range of highly contrasted moods, colour, and tonality.*

This dissertation consists of three episodes. Episode I is a paper-based—otherwise known as cumulative—episode containing chapters that provide insight in advanced DSL embedding techniques for functional programming (FP) languages. The chapters can be read independently of each other. Episode II is a monograph showing mTask, a TOP DSL for the IoT. Hence, the chapters in this episode are best read in order. It introduces IoT edge device programming, shows the complete mTask language, provides details on how mTask is integrated with iTask, shows
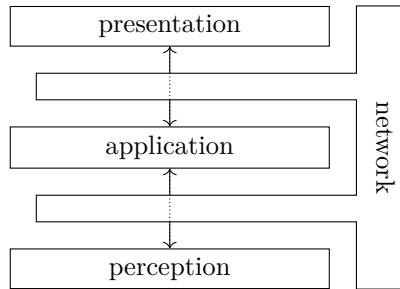
Figure 1.1: A layered IoT architecture.

how the byte code compiler is implemented, presents a guide for green computing with mTask, and ends with a conclusion and overview of future and related work. Episode III consists of a single chapter that is based on a journal article. The chapter provides a qualitative and quantitative comparison of traditional tiered IoT programming and tierless programming using a real-world application. The chapter is readable independently.

The following sections in this prelude provide background material on the IoT, DSLs, and TOP after which a detailed overview of the contributions is presented.

1.2   Internet of things

The IoT is growing rapidly, and it is changing the way people and machines interact with each other and the world. While the term IoT briefly gained interest around 1999 to describe the communication of radio-frequency identification (RFID) devices (Ashton, 1999, 2009), it probably already popped up halfway the eighties in a company speech by Lewis (1985):

> *The internet of things, or IoT, is the integration of people, processes and technology with connectable devices and sensors to enable remote monitoring, status, manipulation and evaluation of trends of such devices.*

Much later, CISCO stated that the IoT started when there were as many connected devices as there were people on the globe, i.e. around 2008 (Evans, 2011). Today, IoT is the term for a system of devices that sense the environment, act upon it, and communicate with each other and the world they operate in. These connected devices are already in households all around us in the form of smart electricity meters, fridges, phones, watches, home automation, &c.

When describing IoT systems, a tiered—or layered—architecture is often used for compartmentalisation. The number of tiers depends on the required complexity of the model. For the intents and purposes of this thesis, the layered architecture as shown in figure 1.1 is used.

To explain the tiers, an example IoT application—home automation—is dissected. Closest to the end-user is the presentation layer. This layer provides the interface between the user and the IoT system. In home automation this may be a web interface, an app used on a phone, or wall-mounted tablet to interact with edge devices and view sensor data.

The application layer is the core of the system. It provides the application programming interfaces (APIs), data interfaces, data storage processing, and data processing of IoT systems. A cloud server or local server provides this layer in a typical home automation application.

The perception layer—also called edge layer—collects the data and interacts with the environment. It consists of edge devices such as microcontrollers equipped with various sensors and actuators. In home automation this layer consists of all devices hosting sensors and actuators such as smart light bulbs, actuators to open doors, or temperature and humidity sensors.

All layers are connected using the network layer. In some applications this is implemented using conventional networking techniques such as Wi-Fi or Ethernet. However, network technology that is tailored to the needs of the specific interconnection between two layers is increasingly popular. Examples of this are BLE, LoRa, ZigBee, and LTE-M as a communication protocol for connecting the perception layer to the application layer using IoT transport protocols such as MQTT. Furthermore, protocols such as HTTP, AJAX, and WebSocket are designed for the use in web applications and connect the presentation layer to the application layer.

Across the layers, the devices are a large heterogeneous collection of different platforms, protocols, paradigms, and programming languages. As a result, impedance problems or semantic friction occurs between layers and the maintainability is severely hampered (Ireland et al., 2009). Even more so, the perception layer often is a heterogeneous collection of microcontrollers in itself, each having their own peculiarities, programming language of choice, and hardware interfaces. As edge hardware needs to be cheap, small scale, and energy efficient, the microcontrollers used to power them do not have a lot of computational power, only a smidge of memory, and little communication bandwidth. Typically, these devices are unable to run a full-fledged general-purpose OS. Rather they employ compiled firmware written in imperative languages that combines all tasks on the device in a single program. While devices are getting a bit faster, smaller, and cheaper, they keep these properties to an extent. For example, more powerful microcontrollers are capable of running real-time OSs (RTOSs), but this still requires a lot of resources and fixes the programs at compile time. As a consequence, the flexibility is greatly reduced for dynamic systems in which tasks are created on the fly, executed on demand, require parallel execution, or have dynamic scheduling behaviour. As program memory is mostly flash-based and only lasts a couple of thousands of writes before it wears out, it is not suitable for repeated reconfiguring and reprogramming.

Memory wear problems can be mitigated by dynamically sending code to be interpreted to the microcontroller. With interpretation, a specialised interpreter is flashed in the program memory once it receives the program code to execute at
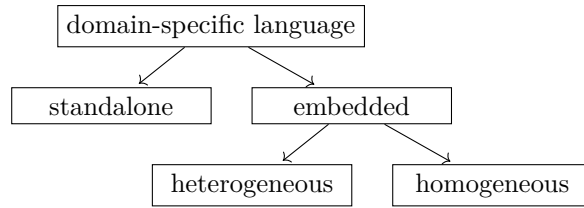
Figure 1.2: A hyponymy of DSLs (adapted from Verna (2013, p. 2)).

run time. Therefore, as the programs are not stored in the flash memory, it does not wear out. It is challenging to create interpreters for small edge devices due to the severe hardware restrictions. This dissertation describes a DSL that includes the high-level programming concepts of TOP, while it can be executed on edge devices with very limited hardware requirements. It does so by compiling the DSL to byte code that is executed in a feather-light domain-specific OS.

1.3   Domain-specific languages

Programming languages can be divided into two categories: DSLs and general-purpose languages (GPLs) (Fowler, 2010). Where GPLs are not made with a demarcated area in mind, DSLs are tailor-made for a specific domain. Writing idiomatic domain-specific code in a DSL is easier and requires less GPL knowledge for a domain expert. This does come at the cost of the DSL being sometimes less expressive to an extent that it may not even be Turing complete. DSLs come in two main flavours: standalone and embedded (section 1.3.1).[1] Standalone languages are languages for which the complete toolchain has been developed, just as for any other GPL. Embedded languages piggyback on an existing language, they are defined in terms of their host language. Embedded domain-specific languages (eDSLs) can further be classified into heterogeneous and homogeneous languages (section 1.3.2). In homogeneous languages all components are integrated whereas in heterogeneous DSLs, some parts are agnostic of the other systems, e.g. a DSL that generates code for execution on a totally different system. This hyponymy is shown in figure 1.2.

1.3.1   Standalone and embedded

DSLs were historically created as standalone languages, meaning that all machinery is developed solely for the language. The advantage of this approach is that the language designer is free to define the syntax and type system of the language as they wish, not being restricted by any constraint. Unfortunately it also means that they need to develop a compiler or interpreter, and all the scaffolding for the language, making standalone DSLs costly to create. Examples of standalone DSLs are TEX, make, yacc, XML, SQL, &c.

---

[1]Standalone and embedded are also called external and internal respectively.

The dichotomous approach to standalone DSLs is embedding the DSL in a host language, i.e. eDSLs (Hudak, 1998). By defining the language as constructs in the host language, much of the machinery is inherited (Krishnamurthi, 2001). This greatly reduces the cost of creating embedded languages and shields the user from having to learn the host language and toolchain. However, there are two sides to this coin. If the syntax of the host language is not very flexible, the syntax of the DSL can become clumsy. Furthermore, DSL errors shown to the programmer may be larded with host language errors, making it difficult for a non-expert of the host language to work with the DSL. FP languages are especially suitable for hosting embedded DSLs. They offer tooling for building abstraction levels by a strong and versatile type system, minimal but flexible syntax, and referential transparency.

1.3.2   Heterogeneity and homogeneity

Tratt (2008) applies a notion from metaprogramming (Sheard, 2001) to eDSLs to define homogeneity and heterogeneity of eDSLs as follows:

> *A homogeneous system is one where all the components are specifically designed to work with each other, whereas in heterogeneous systems at least one of the components is largely, or completely, ignorant of the existence of the other parts of the system.*

Homogeneous eDSLs are languages that are solely defined as an extension to their host language. They often restrict features of the host language to provide a safer interface or capture an idiomatic pattern in the host language for reuse. The difference between a library and a homogeneous eDSLs is not always clear. Examples of homogeneous eDSLs are libraries such as ones for sets, regions, but also more complex tasks such as graphical user interfaces (GUIs).

On the other hand, heterogeneous eDSLs are languages that are not executed in the host language. For example, Elliott et al. (2003) describe the language Pan, for which the final representation in the host language is a compiler that will, when executed, generate code for a completely different target platform.

Both iTask and mTask are eDSLs. Programs written in iTask run in the host language, and hence iTask is a homogeneous DSL. Tasks written using mTask are dynamically compiled to byte code for an edge device, making it a heterogeneous DSL. The interpreter running on the edge device has no knowledge of the higher level task specification. It just interprets the byte code it was sent and takes care of the communication.

1.4   Task-oriented programming

TOP is a declarative programming paradigm for modelling interactive systems (Plasmeijer et al., 2012). Instead of dividing problems into layers, TOP deals with separation of concerns in a novel way. This approach to software development is called task-oriented software development (TOSD) (Stutterheim et al., 2018).
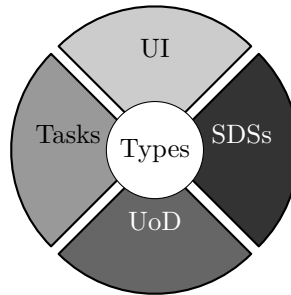
Figure 1.3: Separation of concerns in TOSD (adapted from (Stutterheim et al., 2018, p. 20)).

Types: As can be seen from figure 1.3, types are the pivotal component in TOP. From the data types, utilising various *type-parametrised* concepts, all other aspects are handled automatically. Hence, all other components arise from and depend on the types in the program.

Tasks: In TOP languages, tasks are the basic building blocks. A task is an abstract representation of a piece of work that needs to be done. It provides an abstraction over work in the real world. The nature of tasks makes them observable during execution. It is possible to observe the current task value act upon it, e.g. taking a partial result as good enough, or by starting new tasks. Examples of tasks are filling forms, sending emails, reading sensors or even doing physical tasks. Just as with real-life tasks, multiple tasks can be combined in various ways such as in parallel or in sequence to form workflows. Such combination operators are called task combinators.

Shared data sources (SDSs): Tasks mainly communicate using their observable task values. However, some collaboration patterns are more easily expressed by tasks that share common data. SDSs fill this gap, they offer a safe abstraction over any data. An SDS can represent typed data stored in a file, a chunk of memory, a database, *&c.* SDSs can also represent external impure data such as the time, random numbers or sensor data. In many TOP languages, combinators are available to filter, combine, transform, and focus SDSs.

User interface (UI): The UI of the system is automatically generated from the structural representation of the types. Though, practical TOP systems allow tweaking afterwards to suit the specific needs of the application.

Universe of discourse (UoD): The UoD is explicitly and separately modelled by the data types and relations that exist in the functions of the host language.

Figure 1.3 differs from the presented IoT architecture shown in figure 1.1 because it represents different concepts. The IoT architecture is an execution architecture whereas TOSD is a software development model. E.g. from a software development perspective, a task is a task, whether it is executed on a microcontroller, a server, or a client. Only when a task is executed, the location of the execution becomes important, but this is taken care of by the system. Some concepts from the TOSD

model can be mapped upon the IoT architecture in two ways. Firstly, edge devices can be seen as simple resources, thus accessed through SDSs. The second view is that edge devices contain miniature TOP systems in itself. The individual components in the miniature systems, the tasks, the SDSs, are, in the eventual execution, connected to the main system.

1.4.1 The iTask system

The concept of TOP originated from the iTask framework, a declarative TOP language for defining interactive distributed web applications. The iTask system is implemented as an eDSL in the programming language Clean (Plasmeijer et al., 2007b, 2012).[2] It has been under development for over fifteen years and has proven itself through use in industry. For example, it is the main language of VIIA, an advanced application for monitoring coasts (TOP Software, 2023). Browsers are powering iTask's presentation layer. The browser runs the actual iTask code using an interpreter that operates on Clean's intermediate language ABC (Staps et al., 2019). It is built on top of standard web techniques such as JavaScript, HTML, and CSS. From the structural properties of the data types and the current status of the work to be done, the UI and all interaction is automatically generated.

Tasks in iTask have either *no value*, an *unstable* or a *stable* task value. For example, an editor for filling in a form initially has no value. Once the user enters a complete value, its value becomes an unstable value. It can still be changed or even reverted to no value by emptying the editor again. Only when for example a continue button is pressed, a task value becomes stable, fixing its value. The allowed task value transitions are shown in figure 1.4.

As an example, listing 1.1 and figure 1.5 show the code and UI for an interactive to-do list application. The user modifies a shared to-do list through an editor directly or using some predefined actions. Furthermore, in parallel, the length of the list is shown to demonstrate SDSs. Using iTask, complex collaborations of users and tasks are described on a high level.

From the data type definitions (line 1), using generic programming (line 2), the UIs for the data types are automatically generated. Then, using the parallel task combinator (-||) the task for updating the to-dos (line 8) and the task for viewing the length are combined (line 9, shown as *Length: 2* in the bottom of the figure). This particular parallel combinator uses the result of the left-hand side task. Both tasks operate on the to-do SDS (line 5). The task for updating the to-do list is an editor (line 12) combined using a step combinator (lines 13 to 15). The actions either change the value, sorting or clearing it, or terminate the task by returning the current value of the SDS, visualised as three buttons on the bottom right of the UI. Special combinators (e.g. @>> at line 12) are used to tweak the UI and display informative labels.

---

[2]Appendix A contains a guide for Clean tailored to Haskell programmers.
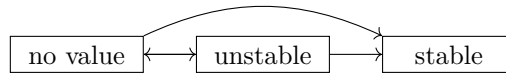
Figure 1.4: Transition diagram for task values in iTask.

```clean
1  :: ToDo = {description :: String, date :: Date}
2  derive class iTask ToDo
3
4  todos :: SimpleSDSLens [ToDo]
5  todos = sharedStore "todos" []
6
7  toDoTask :: Task [ToDo]
8  toDoTask = upToDos
9    -|| viewSharedInformation [ViewAs length] todos <<@ Label "Length"
10
11 upToDos :: Task [ToDo]
12 upToDos = updateSharedInformation [] todos <<@ Title "My todo-list"
13    >>* [ OnAction (Action "Sort") (hasValue \_→upd sort todos >-| upToDos)
14        , OnAction (Action "Clear") (always (set [] todos >-| upToDos))
15        , OnAction (Action "Quit") (always (get todos))
16        ]
17 where sort list = sortBy (\x y→x.ToDo.date < y.ToDo.date) list
```

Listing (Clean) 1.1: The code for a shared to-do list in iTask.
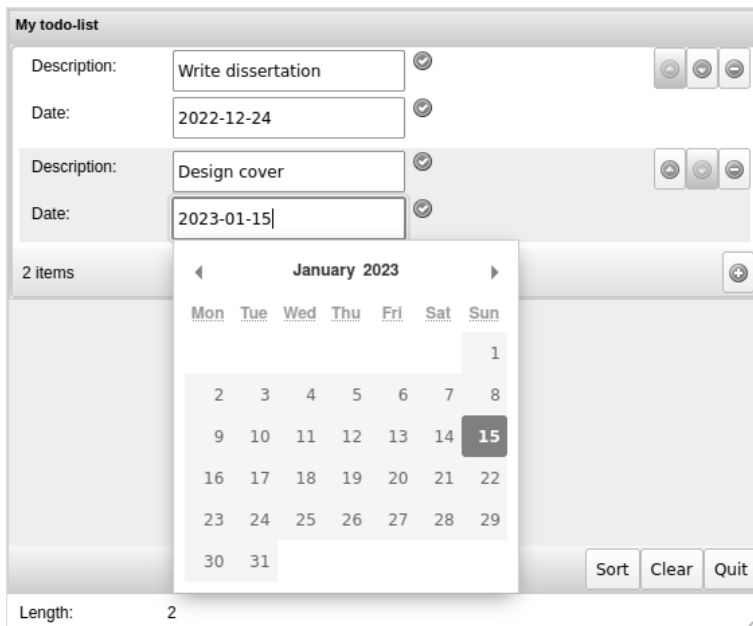
Figure 1.5: The UI for the shared to-do list in iTask.

1.4.2   The mTask system

The work for IoT edge devices can often be succinctly described by TOP programs. Software on microcontrollers is usually composed of smaller basic tasks, is interactive, and shares data with other components or the server. The iTask system seems an obvious candidate for bringing TOP to IoT edge devices. However, an iTask application contains many features that are not needed on *edge devices* such as higher-order tasks, support for a distributed architecture, a multi-user web server, and facilities to generate GUIs for any user-defined type. Furthermore, IoT edge devices are in general not powerful enough to run or interpret Clean/ABC code, they just lack the processor speed and memory. To bridge this gap, mTask is developed, a domain-specific TOP system for IoT edge devices that is integrated in iTask (Koopman et al., 2018). The iTask language abstracts away from details such as user interfaces, data storage, client-side platforms, and persistent workflows. On the other hand, mTask offers abstractions for edge layer-specific details such as the heterogeneity of architectures, platforms, and frameworks; peripheral access; task scheduling; and lowering energy consumption.

The mTask system is seamlessly integrated with iTask. Tasks in mTask are integrated in such a way that they function as regular iTask tasks. Furthermore, SDSs on the device can proxy iTask SDSs. Using mTask, the programmer can define all layers of an IoT system as a single declarative specification. The mTask language is written in Clean as a multi-view eDSL and hence there are multiple interpretations possible. This thesis mostly discusses the byte code compiler. From an mTask task constructed at run time, a compact binary representation of the work that needs to be done is compiled. While the byte code for mTask is generated at run time, the type system of the host language Clean prevents type errors in the generated code. This byte code is then sent to a device that running the mTask run-time system (RTS). This feather-light domain-specific OS is written in portable C with a minimal device specific interface and it executes the tasks using interpretation and rewriting.

To illustrate iTask/mTask, an example application is shown. It is an interactive program for blinking a LED on the microcontroller at a certain frequency that can be set and updated at run time. Listing 1.2 and figure 1.6 show the iTask part of the code and a screenshot. Using `enterInformation`, the connection specification of the TCP device is queried through a web editor (line 2 and figure 1.6a). Line 3 defines an SDS to communicate the blinking interval between the server and the edge device. The mTask device is connected using `withDevice` at line 4. Once connected, the `intBlink` task is sent to the device (line 5) and, in parallel, a web editor is shown that updates the value of the interval SDS (line 6 and figure 1.6b). To allow terminating of the task, the iTask task ends with a sequential operation that returns a constant value when the button is pressed, making the task stable.

The `intBlink` task (listing 1.3) is the mTask part of the application. It blinks an LED on the edge device with the delay that is dynamically adjustable by the user via an iTask editor in the browser. It has its own tasks, SDSs, and UoD. This task first defines general-purpose input/output (GPIO) pin 13 to be of the output type (line 13). Then the iTask SDS is lifted to an mTask SDS (line 14), enabling

```
1  interactiveBlink :: Task ()
2  interactiveBlink = enterDevice
3      >>? \spec→withShared 500 \iInterval→
4          withDevice spec \dev→
5              liftmTask (intBlink iInterval) dev
6          -|| (Hint "Interval (ms)" @>> updateSharedInformation [] iInterval)
7          >>* [OnAction (Action "Stop") (always (return ()))]
8
9  enterDevice :: Task TCPSettings
10 enterDevice = enterInformation [] <<@ Hint "Enter connection info"
```

Listing (Clean) 1.2: The iTask code for the interactive blinking application.

(a) Device selection.   (b) Changing the interval.

Figure 1.6: The UI for the interactive blink application in mTask.

the machinery to keep the SDS in sync both on the device and the server. The main expression of the program calls the `blink` function with an initial state. This function on lines 15 to 19 first reads the interval SDS, waits the specified delay, writes the state to the GPIO pin, and calls itself recursively using the inverse of the state in order to run continuously. The `>>|.` operator denotes the sequencing of tasks in mTask.

```
11 intBlink :: (Shared sds Int) → Main (MTask v Int) | mtask v & ···
12 intBlink iInterval =
13        declarePin D2 PMOutput \ledPin→
14        lowerSds \mInterval = iInterval
15    In fun \blink = (\st→
16              getSds mInterval
17         >>=. \i→delay i
18         >>|. writeD ledPin st
19         >>|. blink (Not st))
20    In {main = blink true}
```

Listing (Clean) 1.3: The mTask code for the interactive blinking application.

1.4.3 Other TOP languages

While iTask conceived TOP, it is no longer the only TOP system. Some TOP languages were created to fill a gap encountered in practice. Toppyt (Lijnse, 2022) is a general purpose TOP language written in Python used to host frameworks for modelling command & control systems. The hTask system is a TOP system written in Haskell used as a vessel for experimenting with asynchronous SDSs (Lubbers, 2022b). Furthermore, some TOP systems arose from Master's and Bachelor's thesis projects. For example, μTask (Piers, 2016), a TOP language for modelling non-interruptible embedded systems in Haskell, and LTasks (van Gemert, 2022), a TOP language written in the dynamically typed programming language Lua. Finally, there are TOP languages with strong academic foundations. TopHat is a fully formally specified TOP language designed to capture the essence of TOP (Steenvoorden et al., 2019). Such a formal specification allows for symbolic execution, hint generation, but also the translation to iTask for actually performing the work (Steenvoorden, 2022).

1.4.4 Tierless programming

Both iTask and iTask/mTask are so called tierless systems. Tierless programming is an entirely different development paradigm compared to traditional, tiered, programming. In tiered programming, every component/tier is separately developed, possibly in different programming languages and programming paradigms, and integrated in the system as a whole. On the one hand, it is an advantage to be able to choose the most suitable programming language for the specific tier. But on the other, it increases the amount of integration work that needs to be done and it may increase the semantic friction between the tiers. In contrast, tierless programming languages synthesise all tiers of a software stack from a single high-level specification. Hence, reducing the semantic friction, increasing the maintainability costs, and reducing the possibility for runtime errors. The term tierless programming originated from the web programming system Links (Cooper et al., 2007). In Links, code for each tier simultaneously checked by the compiler, and compiled to HTML and JavaScript for the web client and to SQL on the server to interact with the database system. The iTask system is a tierless system taking care of both the presentation and application layer (see figure 1.1). When iTask is used in conjunction with mTask, all layers of an IoT system can be programmed from a single source and hence they are a tierless IoT system.

1.5   Contributions

This section provides a thorough overview of the relation between the scientific publications and the contents of this thesis.

1.5.1   Episode I: Étude — Domain-Specific Languages

The mTask system is an eDSL and during the development of it, several novel basal techniques for embedding DSLs in FP languages were found. This paper-based episode is based on the following papers:

1. *Deep Embedding with Class* (Lubbers, 2022a) is the basis for chapter 2. It shows a novel deep embedding technique for DSLs where the resulting language is extendible both in constructs and in interpretation just using type classes and existential data types. The related work section is updated with the research found after publication. Section 2.A was added after publication and contains a (yet) unpublished extension of the embedding technique for reducing the required boilerplate at the cost of requiring some advanced type system extensions. The paper was published at the *International Symposium on Trends in Functional Programming (TFP)* 2022 in Krakow, Poland (moved to online).

2. *First-Class Data Types in Shallow Embedded Domain-Specific Languages* (Lubbers, Koopman and Plasmeijer, 2023a) is the basis for chapter 3. It shows how to inherit data types from the host language in eDSLs using meta-programming by providing a proof-of-concept implementation using Haskell's metaprogramming system: Template Haskell. The chapter also serves as a gentle introduction to, and contains a thorough literature study on Template Haskell. The paper was published at the *Symposium on Implementation and Application of Functional Languages (IFL)* 2022 in Copenhagen, Denmark.

Contribution: The papers are written by me, there were weekly meetings with co-authors in which we discussed and refined the ideas.

1.5.2   Episode II: Orchestrating the Internet of Things using Task-Oriented Programming

This episode is a monograph that shows the design, properties, implementation and usage of the mTask system and TOP for the IoT. It is compiled from the following publications:

3. *A Task-Based DSL for Microcomputers* (Koopman, Lubbers and Plasmeijer, 2018) is the initial TOP/mTask paper. It provides an overview of the initial TOP mTask language and shows first versions of some interpretations. The paper was published at the *International Workshop on Real World Domain Specific Languages (RWDSL)* 2018 in Vienna, Austria.

4. *Task Oriented Programming for the Internet of Things* (Lubbers, Koopman and Plasmeijer, 2018)[3] shows how a simple imperative variant of mTask was integrated with iTask. While the language differs a lot from the current version, the integration mechanism is still used. The paper was published at the *Symposium on Implementation and Application of Functional Languages (IFL)* 2018 in Lowell, MA, USA.

---

[3]This work is an extension of my Master's thesis (Lubbers, 2017).

5. *Multitasking on Microcontrollers using Task Oriented Programming* (Lubbers, Koopman and Plasmeijer, 2019)[4] is a short paper on the multitasking capabilities of mTask comparing it to traditional multitasking methods for Arduino.
   The paper was published at the *COnference on COmposability, COmprehensibility and COrrectness of Working Software (4COWS)* 2019 in Opatija, Croatia.

6. *Simulation of a Task-Based Embedded Domain Specific Language for the Internet of Things* (Koopman, Lubbers and Plasmeijer, 2023)[4] are the revised lecture notes for a course on the mTask simulator provided at the 2018 3COWS winter school in Košice, Slovakia, January 22–26, 2018.

7. *Writing Internet of Things Applications with Task Oriented Programming* (Lubbers, Koopman and Plasmeijer, 2023b)[4] are the revised lecture notes from a course on programming IoT systems using mTask provided at the 2019 3COWS summer school in Budapest, Hungary, June 17–21, 2019.

8. *Interpreting Task Oriented Programs on Tiny Computers* (Lubbers, Koopman and Plasmeijer, 2021) shows an implementation of the byte code compiler and RTS of mTask. The paper was published at the *Symposium on Implementation and Application of Functional Languages (IFL)* 2019 in Singapore.

9. *Reducing the Power Consumption of IoT with Task-Oriented Programming* (Crooijmans, Lubbers and Koopman, 2022) shows how to create a scheduler so that devices running mTask tasks can go to sleep more automatically and how interrupts are incorporated in the language. The paper was published at the *International Symposium on Trends in Functional Programming (TFP)* 2022 in Krakow, Poland (moved to online).

10. *Green Computing for the Internet of Things* (Lubbers and Koopman, 2022)[5] are the revised lecture notes from a course on sustainable IoT programming with mTask provided at the 2022 SusTrainable summer school in Rijeka, Croatia, July 4–8, 2022.

Contribution: The original mTask language, and their initial interpretations were developed by Pieter Koopman and Rinus Plasmeijer. I extended the language, developed the byte code interpreter, the integration with iTask, and the RTS. The papers of which I am first author are solely written by me, there were weekly meetings with the co-authors in which we discussed and refined the ideas.

1.5.3   Episode III: Tiered versus Tierless Programming

Episode III is based on a journal paper that quantitatively and qualitatively compares traditional IoT architectures with TOP IoT architectures.

11. *Could Tierless Programming Reduce IoT Development Grief?* (Lubbers, Koopman, Ramsingh, Singer and Trinder, 2023c) is an extended version of paper 12. It compares programming traditional tiered architectures to tierless architectures by illustrating a qualitative and a quantitative four-way comparison of a smart-campus application. The paper was published in the *ACM Transactions on Internet of Things (TIoT)* journal.

12. *Tiered versus Tierless IoT Stacks: Comparing Smart Campus Software Architectures* (Lubbers, Koopman, Ramsingh, Singer and Trinder, 2020)[6] compares traditional tiered programming to tierless architectures by comparing two implementations of a smart-campus application. The paper was published in the *International Conference on the Internet of Things (IoT)* 2020 in Malmö, Sweden (moved to online).

Contribution: Writing the paper was performed by all authors. I created the server application, the Clean/iTask/mTask implementation (CWS), and the Clean/iTask implementation (CRS); Adrian Ramsingh created the MicroPython implementation (PWS); the original Python implementation (PRS) and the server application were created by Hentschel et al. (2016).

Episode I: Étude

Domain-Specific Languages

Chapter 2

Deep embedding with class

The two flavours of DSL embedding are shallow and deep embedding. In functional languages, shallow embedding models the language constructs as functions in which the semantics are embedded. Adding semantics is therefore cumbersome while adding constructs is a breeze. Upgrading the functions to type classes lifts this limitation to a certain extent.

Deeply embedded languages represent their language constructs as data and the semantics are functions on it. As a result, the language constructs are embedded in the semantics, hence adding new language constructs is laborious where adding semantics is trouble free.

This chapter shows that by abstracting the semantics functions in deep embedding to type classes, it is possible to easily add language constructs as well. So-called classy deep embedding results in DSLs that are extensible both in language constructs and in semantics while maintaining a concrete abstract syntax tree. Additionally, little type-level trickery or complicated boilerplate code is required to achieve this.

2.1 Introduction

The two flavours of DSL embedding are deep and shallow embedding (Boulton et al., 1992). In FP languages, shallow embedding models language constructs as functions in the host language. As a result, adding new language constructs—extra functions—is easy. However, the semantics of the language is embedded in these functions, making it troublesome to add semantics since it requires updating all existing language constructs.

On the other hand, deep embedding models language constructs as data in the host language. The semantics of the language are represented by functions over the

data. Consequently, adding new semantics, i.e. novel functions, is straightforward. It can be stated that the language constructs are embedded in the functions that form a semantics. If one wants to add a language construct, all semantics functions must be revisited and revised to avoid ending up with partial functions.

This juxtaposition has been known for many years (Reynolds, 1978) and discussed by many others (Krishnamurthi et al., 1998) but most famously dubbed the *expression problem* by Wadler (1998):

> The *expression problem* is a new name for an old problem. The goal is to define a data type by cases, where one can add new cases to the data type and new functions over the data type, without recompiling existing code, and while retaining static type safety (e.g., no casts).

In shallow embedding, abstracting the functions to type classes disentangles the language constructs from the semantics, allowing extension both ways. This technique is dubbed tagless-final embedding (Carette et al., 2009), nonetheless it is no silver bullet. Some semantics that require an intensional analysis of the syntax tree, such as transformation and optimisations, are difficult to implement in shallow embedding due to the lack of an explicit data structure representing the abstract syntax tree. The semantics of the DSL have to be combined and must hold some kind of state or context, so that structural information is not lost (Kiselyov, 2012).

2.1.1   Research contribution

This chapter shows how to apply the technique observed in tagless-final embedding to deep embedding. The presented basic technique, christened *classy deep embedding*, does not require advanced type system extensions to be used. However, it is suitable for type system extensions such as generalised ADTs (GADTs). While this chapter is written as a literate Haskell (Peyton Jones, 2003) program using some minor extensions provided by Glasgow Haskell compiler (GHC) (GHC Team, 2021b), the idea is applicable to other languages as well.[1]

2.2   Deep embedding

Let us consider the language of literal integers and addition.In deep embedding, terms in the language are represented by data in the host language. Hence, defining the constructs is as simple as creating the following algebraic data type.[2]

```
data Expr₀
    = Lit₀ Int
    | Add₀ Expr₀ Expr₀
```

---

[1]Lubbers, M. (2021): Literate Haskell/lhs2TₑX source code of the paper "Deep Embedding with Class": TFP 2022. Zenodo. https://doi.org/10.5281/zenodo.6650880.

[2]All data types and functions are subscripted to indicate the evolution. When definitions are omitted for version *n*, version *n - 1* is assumed.

Semantics are defined as functions on the $\mathtt{Expr_0}$ data type. For example, a function transforming the term to an integer—an evaluator—is implemented as follows.

```
eval₀ :: Expr₀  →  Int
eval₀ (Lit₀ e)    = e
eval₀ (Add₀ e₁ e₂) = eval₀ e₁ + eval₀ e₂
```

Adding semantics—e.g. a printer—just means adding another function while the existing functions remain untouched, i.e. the key property of deep embedding. The following function, transforming the $\mathtt{Expr_0}$ data type to a string, defines a simple printer for our language.

```
print₀ :: Expr₀  →  String
print₀ (Lit₀ v)    = show v
print₀ (Add₀ e₁ e₂) = "(" ++ print₀ e₁ ++ "−" ++ print₀ e₂ ++ ")"
```

While the language is concise and elegant, it is not very expressive. Traditionally, extending the language is achieved by adding a case to the $\mathtt{Expr_0}$ data type. So, adding subtraction to the language results in the following revised data type.

```
data Expr₀
    = Lit₀ Int
    | Add₀ Expr₀ Expr₀
    | Sub₀ Expr₀ Expr₀
```

Extending the DSL with language constructs exposes the Achilles' heel of deep embedding. Adding a case to the data type means that all semantics functions have become partial and need to be updated to be able to handle this new case. This does not seem like an insurmountable problem, but it does pose a problem if either the functions or the data type itself are written by others or are contained in a closed library.

2.3   Shallow embedding

Conversely, let us see how this would be done in shallow embedding. First, the data type is represented by functions in the host language with embedded semantics. Therefore, the evaluators for literals and addition both become a function in the host language as follows.

```
type Semₛ = Int

litₛ :: Int  →  Semₛ
litₛ i = i

addₛ :: Semₛ  →  Semₛ  →  Semₛ
addₛ e₁ e₂ = e₁ + e₂
```

Adding constructions to the language is done by adding functions. Hence, the following function adds subtraction to our language.

```
subₛ :: Semₛ → Semₛ → Semₛ
subₛ e₁ e₂ = e₁ - e₂
```

Adding semantics on the other hand—e.g. a printer—is not that simple because the semantics are part of the functions representing the language constructs. One way to add semantics is to change all functions to execute both semantics at the same time. In our case this means changing the type of `Semₛ` to be `(Int, String)` so that all functions operate on a tuple containing the result of the evaluator and the printed representation at the same time. Alternatively, a single semantics can be defined that represents a fold over the language constructs (Gibbons and Wu, 2014), delaying the selection of semantics to the moment the fold is applied.

2.3.1   Tagless-final embedding

Tagless-final embedding overcomes the limitations of standard shallow embedding. To upgrade to this embedding technique, the language constructs are changed from functions to type classes. For our language this results in the following type class definition.

```
class Exprₜ s where
    litₜ :: Int → s
    addₜ :: s → s → s
```

Semantics become data types implementing these type classes, resulting in the following instance for the evaluator.[3]

```
newtype Evalₜ = Eₜ Int

instance Exprₜ Evalₜ where
    litₜ v               = Eₜ v
    addₜ (Eₜ e₁) (Eₜ e₂) = Eₜ (e₁ + e₂)
```

Adding constructs—e.g. subtraction—just results in an extra type class and corresponding instances.

```
class Subₜ s where
    subₜ :: s → s → s

instance Subₜ Evalₜ where
    subₜ (Eₜ e₁) (Eₜ e₂) = Eₜ (e₁ - e₂)
```

Finally, adding semantics such as a printer for the language is achieved by providing a data type representing the semantics accompanied by instances for the language constructs.

```
newtype Printerₜ = Pₜ String
```

---

[3]In this case **newtype**s are used instead of regular **data** declarations. A **newtype** is a special data type with a single constructor containing a single value only to which it is isomorphic. It allows the programmer to define separate class instances that the instances of the isomorphic type without any overhead. During compilation the constructor is completely removed (Peyton Jones, 2003, section 4.2.3).

```
instance Expr_t Printer_t where
    lit_t i                = P_t (show i)
    add_t (P_t e_1) (P_t e_2) = P_t ("(" ++ e_1 ++ "+" ++ e_2 ++ ")")

instance Sub_t Printer_t where
    sub_t (P_t e_1) (P_t e_2) = P_t ("(" ++ e_1 ++ "−" ++ e_2 ++ ")")
```

2.4   Lifting the interpretations

Let us rethink the deeply embedded DSL design. Remember that in shallow embedding, the semantics are embedded in the language construct functions. Obtaining extensibility both in constructs and semantics was accomplished by abstracting the semantics functions to type classes, making the constructs overloaded in the semantics. In deep embedding, the constructs are embedded in the semantics functions instead. So, let us apply the same technique, i.e. make the semantics overloaded in the language constructs by abstracting the semantics functions to type classes. The same effect may be achieved when using similar techniques such as explicit dictionary passing or ML style modules. In our language this results in the following type class.

```
class Eval_1 v where
    eval_1 :: v → Int

data Expr_1
    = Lit_1 Int
    | Add_1 Expr_1 Expr_1
```

Implementing the semantics type class instances for the `Expr_1` data type is an elementary exercise. By a copy-paste and some modifications, we come to the following implementation.

```
instance Eval_1 Expr_1 where
    eval_1 (Lit_1 v)     = v
    eval_1 (Add_1 e_1 e_2) = eval_1 e_1 + eval_1 e_2
```

Subtraction can now be defined in a separate data type, leaving the original data type intact. Instances for the additional semantics can now be implemented separately as instances of the type classes.

```
data Sub_1 = Sub_1 Expr_1 Expr_1

instance Eval_1 Sub_1 where
    eval_1 (Sub_1 e_1 e_2) = eval_1 e_1 − eval_1 e_2
```

2.5   Existential data types

The astute reader might have noticed that we have dissociated ourselves from the original data type. It is only possible to create an expression with a subtraction on

the top level. The recursive knot is left untied and as a result, Sub_1 can never be reached from an $Expr_1$.

Luckily, we can reconnect them by adding a special constructor to the $Expr_1$ data type for housing extensions. It contains an existentially quantified (Mitchell and Plotkin, 1988) type with type class constraints (Läufer, 1994, 1996) for all semantics type classes (GHC Team, 2021b, section 6.4.6) to allow it to house not just subtraction but any future extension.

```
data Expr₂
    =                        Lit₂ Int
    |                        Add₂ Expr₂ Expr₂
    | forall x. Eval₂ x ⇒ Ext₂ x
```

The implementation of the extension case in the semantics type classes is in most cases just a matter of calling the function for the argument as can be seen in the semantics instances shown below.

```
instance Eval₂ Expr₂ where
    eval₂ (Lit₂  v)     = v
    eval₂ (Add₂  e₁ e₂) = eval₂ e₁ + eval₂ e₂
    eval₂ (Ext₂  x)     = eval₂ x
```

Adding language construct extensions in different data types does mean that an extra Ext_2 tag is introduced when using the extension. This burden can be relieved by creating a smart constructor for it that automatically wraps the extension with the Ext_2 constructor so that it is of the type of the main data type.

```
sub₂ :: Expr₂ → Expr₂ → Expr₂
sub₂ e₁ e₂ = Ext₂ (Sub₂ e₁ e₂)
```

In our example this means that the programmer can write[4]:

```
e₂ :: Expr₂
e₂ = Lit₂ 42 `sub₂` Lit₂ 1
```

instead of having to write

```
e₂′ :: Expr₂
e₂′ = Ext₂ (Lit₂ 42 `Sub₂` Lit₂ 1)
```

2.5.1   Unbraiding the semantics from the data

This approach does reveal a minor problem. Namely, that all semantics type classes are braided into our datatypes via the Ext_2 constructor. Say if we add the printer again, the Ext_2 constructor has to be modified to contain the printer type class constraint as well.[5] Thus, if we add semantics, the main data type's type class constraints in the Ext_2 constructor need to be updated. To avoid this, the type classes can be bundled in a type class alias or type class collection as follows.

---

[4]Backticks are used to use functions or constructors in an infix fashion (Peyton Jones, 2003, section 4.3.3).

[5]Resulting in the following constructor: **forall** x.(Eval₂ x, Print₂ x) ⇒ Ext₂ x.

```
class (Eval₂ x, Print₂ x) ⇒ Semantics₂ x

data Expr₂
    =                           Lit₂ Int
    |                           Add₂ Expr₂ Expr₂
    | forall x. Semantics₂ x ⇒ Ext₂ x
```

The class alias removes the need for the programmer to visit the main data type when adding additional semantics. Unfortunately, the compiler does need to visit the main data type again. Some may argue that adding semantics happens less frequently than adding language constructs, but in reality it means that we have to concede that the language is not as easily extensible in semantics as in language constructs. More exotic type system extensions such as constraint kinds (Bolingbroke, 2011; Yorgey et al., 2012) can untangle the semantics from the data types by making the data types parametrised by the particular semantics. However, by adding some boilerplate, even without this extension, the language constructs can be parametrised by the semantics by putting the semantics functions in a data type. First the data types for the language constructs are parametrised by the type variable `d` as follows.

```
data Expr₃ d
    =           Lit₃ Int
    |           Add₃ (Expr₃ d) (Expr₃ d)
    | forall x. Ext₃ (d x) x

data Sub₃ d = Sub₃ (Expr₃ d) (Expr₃ d)
```

The `d` type variable is inhabited by an explicit dictionary for the semantics, i.e. a witness to the class instance. Therefore, for all semantics type classes, a data type is defined which contains the semantics function for the given semantics. This means that for `Eval₃`, a dictionary with the function `EvalDict₃` is defined, a type class `HasEval₃` for retrieving the function from the dictionary and an instance for `HasEval₃` for `EvalDict₃`.

```
newtype EvalDict₃ v = EvalDict₃ (v → Int)

class HasEval₃ d where
    getEval₃ :: d v → v → Int

instance HasEval₃ EvalDict₃ where
    getEval₃ (EvalDict₃ e) = e
```

The instances for the type classes change as well according to the change in the datatype. Given that there is a `HasEval₃` instance for the witness type `d`, we can provide an implementation of `Eval₃` for `Expr₃ d`.

```
instance HasEval₃ d ⇒ Eval₃ (Expr₃ d) where
    eval₃ (Lit₃ v)     = v
    eval₃ (Add₃ e₁ e₂) = eval₃ e₁ + eval₃ e₂
    eval₃ (Ext₃ d x)   = getEval₃ d x
```

```
instance HasEval₃ d ⇒ Eval₃ (Sub₃ d) where
    eval₃ (Sub₃ e₁ e₂) = eval₃ e₁ - eval₃ e₂
```

Because the `Ext₃` constructor from `Expr₃` now contains a value of type `d`, the smart constructor for `Sub₃` must somehow come up with this value. To achieve this, a type class is introduced that allows the generation of such a dictionary.

```
class GDict a where
    gdict :: a
```

This type class has individual instances for all semantics dictionaries, linking the class instance to the witness value. I.e. if there is a type class instance known, a witness value can be conjured using the `gdict` function.

```
instance Eval₃ v ⇒ GDict (EvalDict₃ v) where
    gdict = EvalDict₃ eval₃
```

With these instances, the semantics function can be retrieved from the `Ext₃` constructor and in the smart constructors they can be generated as follows:

```
sub₃ :: GDict (d (Sub₃ d)) ⇒ Expr₃ d → Expr₃ d → Expr₃ d
sub₃ e₁ e₂ = Ext₃ gdict (Sub₃ e₁ e₂)
```

Finally, we reached the end goal, orthogonal extension of both language constructs as shown by adding subtraction to the language, and in language semantics. Adding the printer can now be done without touching the original code as follows. First the printer type class, dictionaries and instances for `GDict` are defined.

```
class Print₃ v where
    print₃ :: v → String

newtype PrintDict₃ v = PrintDict₃ (v → String)

class HasPrint₃ d where
    getPrint₃ :: d v → v → String

instance HasPrint₃ PrintDict₃ where
    getPrint₃ (PrintDict₃ e) = e

instance Print₃ v ⇒ GDict (PrintDict₃ v) where
    gdict = PrintDict₃ print₃
```

Then the instances for `Print₃` of all the language constructs can be defined.

```
instance HasPrint₃ d ⇒ Print₃ (Expr₃ d) where
    print₃ (Lit₃ v)     = show v
    print₃ (Add₃ e₁ e₂) = "(" ++ print₃ e₁ ++ "+" ++ print₃ e₂ ++ ")"
    print₃ (Ext₃ d x)   = getPrint₃ d x
instance HasPrint₃ d ⇒ Print₃ (Sub₃ d) where
    print₃ (Sub₃ e₁ e₂) = "(" ++ print₃ e₁ ++ "−" ++ print₃ e₂ ++ ")"
```

2.6   Transformation semantics

Most semantics convert a term to some final representation and can be expressed just by functions on the cases. However, the implementation of semantics such as transformation or optimisation may benefit from a so-called intentional analysis of the abstract syntax tree. In shallow embedding, the implementation for these types of semantics is difficult because there is no tangible abstract syntax tree. In off-the-shelf deep embedding this is effortless since the function can pattern match on the constructor or structures of constructors.

To demonstrate intensional analyses in classy deep embedding we write an optimiser that removes addition and subtraction by zero. In classy deep embedding, adding new semantics means first adding a new type class housing the function including the machinery for the extension constructor.

```
class Opt₃ v where
    opt₃ :: v → v

newtype OptDict₃ v = OptDict₃ (v → v)

class HasOpt₃ d where
    getOpt₃ :: d v → v → v

instance HasOpt₃ OptDict₃ where
    getOpt₃ (OptDict₃ e) = e

instance Opt₃ v ⇒ GDict (OptDict₃ v) where
    gdict = OptDict₃ opt₃
```

The implementation of the optimiser for the `Expr₃` data type is no complicated task. The only interesting bit occurs in the `Add₃` constructor, where we pattern match on the optimised children to determine whether an addition with zero is performed. If this is the case, the addition is removed.

```
instance HasOpt₃ d ⇒ Opt₃ (Expr₃ d) where
    opt₃ (Lit₃ v)     = Lit₃ v
    opt₃ (Add₃ e₁ e₂) = case (opt₃ e₁, opt₃ e₂) of
        (Lit₃ 0, e₂′   ) → e₂′
        (e₁′,    Lit₃ 0) → e₁′
        (e₁′,    e₂′   ) → Add₃ e₁′ e₂′
    opt₃ (Ext₃ d x)   = Ext₃ d (getOpt₃ d x)
```

Replicating this for the `Opt₃` instance of `Sub₃` seems a clear-cut task at first glance.

```
instance HasOpt₃ d ⇒ Opt₃ (Sub₃ d) where
    opt₃ (Sub₃ e₁ e₂) = case (opt₃ e₁, opt₃ e₂) of
        (e₁′, Lit₃ 0) → e₁′
        (e₁′, e₂′   ) → Sub₃ e₁′ e₂′
```

Unsurprisingly, this code is rejected by the compiler. When a literal zero is matched as the right-hand side of a subtraction, the left-hand side of type `Expr₃` is

returned. However, the type signature of the function dictates that it should be of type Sub_3. To overcome this problem we add a convolution constructor.

2.6.1   Convolution

Adding a loopback case or convolution constructor to Sub_3 allows the removal of the Sub_3 constructor while remaining the Sub_3 type. It should be noted that a loopback case is *only* required if the transformation actually removes tags. This changes the Sub_3 data type as follows.

```
data Sub₄ d
    = Sub₄      (Expr₄ d) (Expr₄ d)
    | SubLoop₄ (Expr₄ d)

instance HasEval₄ d ⇒ Eval₄ (Sub₄ d) where
    eval₄ (Sub₄      e₁ e₂) = eval₄ e₁ - eval₄ e₂
    eval₄ (SubLoop₄ e₁)     = eval₄ e₁
```

With this loopback case in the toolbox, the following Sub instance optimises away subtraction with zero literals.

```
instance HasOpt₄ d ⇒ Opt₄ (Sub₄ d) where
    opt₄ (Sub₄ e₁ e₂) = case (opt₄ e₁, opt₄ e₂) of
        (e₁', Lit₄ 0) → SubLoop₄ e₁'
        (e₁', e₂'    ) → Sub₄ e₁' e₂'
    opt₄ (SubLoop₄ e) =  SubLoop₄ (opt₄ e)
```

2.6.2   Pattern matching

Pattern matching within datatypes and from an extension to the main data type works out of the box. Cross-extensional pattern matching on the other hand—matching on a particular extension—is something that requires a bit of extra care. Take for example negation propagation and double negation elimination. Pattern matching on values with an existential type is not possible without leveraging dynamic typing (Abadi et al., 1991; Baars and Swierstra, 2002). To enable dynamic typing support, the `Typeable` type class as provided by `Data.Dynamic` (GHC Team, 2021a) is added to the list of constraints in all places where we need to pattern match across extensions. As a result, the `Typeable` type class constraints are added to the quantified type variable x of the Ext_4 constructor and to ds in the smart constructors.

```
data Expr₄ d
    =                          Lit₄ Int
    |                          Add₄ (Expr₄ d) (Expr₄ d)
    | forall x. Typeable x ⇒ Ext₄ (d x) x
```

First let us add negation to the language by defining a datatype representing it. Negation elimination requires the removal of negation constructors, so a convolution constructor is defined as well.

```
data Neg₄ d
    = Neg₄      (Expr₄ d)
    | NegLoop₄ (Expr₄ d)

neg₄ :: (Typeable d, GDict (d (Neg₄ d))) ⇒ Expr₄ d → Expr₄ d
neg₄ e = Ext₄ gdict (Neg₄ e)
```

The evaluation and printer instances for the Neg_4 datatype are defined as follows.

```
instance HasEval₄ d ⇒ Eval₄ (Neg₄ d) where
    eval₄ (Neg₄      e) = negate (eval₄ e)
    eval₄ (NegLoop₄ e) = eval₄ e

instance HasPrint₄ d ⇒ Print₄ (Neg₄ d) where
    print₄ (Neg₄      e) = "(∼" ++ print₄ e ++ ")"
    print₄ (NegLoop₄ e) = print₄ e
```

The Opt_4 instance contains the interesting bit. If the sub expression of a negation is an addition, negation is propagated downwards. If the sub expression is again a negation, something that can only be found out by a dynamic pattern match, it is replaced by a $NegLoop_4$ constructor.

```
instance (Typeable d, GDict (d (Neg₄ d)), HasOpt₄ d) ⇒ Opt₄ (Neg₄ d) where
    opt₄ (Neg₄ (Add₄ e₁ e₂))
        = NegLoop₄ (Add₄ (opt₄ (neg₄ e₁)) (opt₄ (neg₄ e₂)))
    opt₄ (Neg₄ (Ext₄ d x))
        = case fromDynamic (toDyn (getOpt₄ d x)) of
            Just (Neg₄ e) → NegLoop₄ e
            _             → Neg₄ (Ext₄ d (getOpt₄ d x))
    opt₄ (Neg₄      e) = Neg₄ (opt₄ e)
    opt₄ (NegLoop₄ e) = NegLoop₄ (opt₄ e)
```

Loopback cases do make cross-extensional pattern matching less modular in general. For example, Ext_4 d ($SubLoop_4$ (Lit_4 0)) is equivalent to Lit_4 0 in the optimisation semantics and would require an extra pattern match. Fortunately, this problem can be mitigated—if required—by just introducing an additional optimisation semantics that removes loopback cases. Luckily, one does not need to resort to these arguably blunt matters often. Dependent language functionality often does not need to span extensions, i.e. it is possible to group them in the same data type.

2.6.3   Chaining semantics

Now that the data types are parametrised by the semantics, a final problem needs to be overcome. The data type is parametrised by the semantics, thus, using multiple semantics, such as evaluation after optimising is not straightforwardly possible. Luckily, a solution is readily at hand: introduce an ad-hoc combination semantics.

```
data OptPrintDict₄ v = OPD₄ (OptDict₄ v) (PrintDict₄ v)
```

```
instance HasOpt₄   OptPrintDict₄ where
    getOpt₄  (OPD₄ v _) = getOpt₄ v
instance HasPrint₄ OptPrintDict₄ where
    getPrint₄ (OPD₄ _ v) = getPrint₄ v

instance (Opt₄ v, Print₄ v) ⇒ GDict ( OptPrintDict₄  v) where
    gdict = OPD₄ gdict gdict
```

And this allows us to write $\texttt{print}_4\ (\texttt{opt}_4\ \texttt{e}_1)$ resulting in `"((~42)+(~38))"` when $\texttt{e}_1$ represents $(\sim (42 + 38)) - 0$ and is thus defined as follows.

```
e₁ :: Expr₄   OptPrintDict₄
e₁ = neg₄ (Lit₄ 42 `Add₄` Lit₄ 38) `sub₄` Lit₄ 0
```

When using classy deep embedding to the fullest, the ability of the compiler to infer very general types expires. As a consequence, defining reuseable expressions that are overloaded in their semantics requires quite some type class constraints that cannot be inferred by the compiler (yet) if they use many extensions. Solving this remains future work. For example, the expression $\sim (42 - 38) + 1$ has to be defined as:

```
e₃ :: (Typeable d, GDict (d (Neg₄ d)), GDict (d (Sub₄ d))) ⇒ Expr₄ d
e₃ = neg₄ (Lit₄ 42 `sub₄` Lit₄ 38) `Add₄` Lit₄ 1
```

2.7   Generalised algebraic data types

GADTs are enriched data types that allow the type instantiation of the constructor to be explicitly defined (Cheney and Hinze, 2003; Hinze, 2003). Leveraging GADTs, deeply embedded DSLs can be made statically type safe even when different value types are supported. Still when GADTs are not supported natively in the language, they can be simulated using embedding-projection pairs or equivalence types (Cheney and Hinze, 2002, section 2.2). Where some solutions to the expression problem do not easily generalise to GADTs (see section 2.8), classy deep embedding does. Generalising the data structure of our DSL is fairly straightforward and to spice things up a bit, we add an equality and boolean negation language construct. To make the existing DSL constructs more general, we relax the types of those constructors. For example, operations on integers now work on all numerals instead. Moreover, the $\texttt{Lit}_g$ constructor can be used to lift values of any type to the DSL domain as long as they have a `Show` instance, required for the printer. Since some optimisations on $\texttt{Not}_g$ remove constructors and therefore use cross-extensional pattern matches, `Typeable` constraints are added to `a`. Furthermore, because the optimisations for $\texttt{Add}_g$ and $\texttt{Sub}_g$ are now more general, they do not only work for `Ints` but for any type with a `Num` instance, the `Eq` constraint is added to these constructors as well. Finally, not to repeat ourselves too much, we only show the parts that substantially changed. The omitted definitions and implementation can be found in section 2.B.

```
data Expr_g d a where
    Lit_g     :: Show a          ⇒ a → Expr_g d a
```

```
    Add_g      :: (Eq a, Num a) ⇒ Expr_g d a → Expr_g d a → Expr_g d a
    Ext_g      :: Typeable x    ⇒ d x → x a → Expr_g d a
data Neg_g  d a where
    Neg_g      :: (Typeable a, Num a) ⇒ Expr_g d a → Neg_g d a
    NegLoop_g ::                              Expr_g d a → Neg_g d a
data Not_g  d a where
    Not_g      :: Expr_g d Bool → Not_g d Bool
    NotLoop_g :: Expr_g d a     → Not_g d a
```

The smart constructors for the language extensions inherit the class constraints of their data types and include a `Typeable` constraint on the `d` type variable for it to be useable in the `Ext_g` constructor as can be seen in the smart constructor for `Neg_g`:

```
neg_g  :: (Typeable d, GDict (d (Neg_g d)), Typeable a, Num a) ⇒
    Expr_g d a → Expr_g d a
neg_g e = Ext_g gdict (Neg_g e)

not_g  :: (Typeable d,  GDict (d (Not_g d))) ⇒
    Expr_g d Bool → Expr_g d Bool
not_g e = Ext_g gdict (Not_g e)
```

Upgrading the semantics type classes to support GADTs is done by an easy textual search and replace. All occurrences of `v` are now parametrised by type variable `a`:

```
class Eval_g  v where
    eval_g  :: v a → a
class Print_g v where
    print_g :: v a → String
class Opt_g   v where
    opt_g   :: v a → v a
```

Now that the shape of the type classes has changed, the dictionary data types and the type classes need to be adapted as well. The introduced type variable `a` is not an argument to the type class, so it should not be an argument to the dictionary data type. To represent this type class function, a rank-2 polymorphic function is needed (GHC Team, 2021b, section 6.4.15)(Odersky and Läufer, 1996). Concretely, for the evaluator this results in the following definitions:

```
newtype EvalDict_g v = EvalDict_g (forall a. v a → a)
class HasEval_g d where
    getEval_g :: d v → v a → a
instance HasEval_g EvalDict_g where
    getEval_g (EvalDict_g e) = e
```

The `GDict` type class is general enough, so the instances can remain the same. The `Eval_g` instance of `GDict` looks as follows:

```
instance Eval_g v ⇒ GDict (EvalDict_g v) where
    gdict = EvalDict_g eval_g
```

Finally, the implementations for the instances can be ported without complication using the optimisation instance of `Not_g`:

```
instance (Typeable d, GDict (d (Notg d)), HasOptg d) ⇒ Optg (Notg d) where
    optg (Notg (Extg d x))
        = case fromDynamic (toDyn (getOptg d x)) :: Maybe (Notg d Bool) of
            Just (Notg e) → NotLoopg e
            _              → Notg (Extg d (getOptg d x))
    optg (Notg e)     = Notg (optg e)
    optg (NotLoopg e) = NotLoopg (optg e)
```

2.8 Related work

Embedded DSL techniques in functional languages have been a topic of research
for many years, thus we do not claim a complete overview of related work.

Clearly, classy deep embedding bears most similarity to the *Datatypes à la
Carte* (Swierstra, 2008). In Swierstra's approach, semantics are lifted to type classes
similarly to classy deep embedding. Each language construct is their own datatype
parametrised by a type parameter. This parameter contains some type level
representation of language constructs that are in use. In classy deep embedding,
extensions only have to be enumerated at the type level when the term is required
to be overloaded, in all other cases they are captured in the extension case. Because
all the constructs are expressed in the type system, nifty type system tricks need
to be employed to convince the compiler that everything is type safe and the class
constraints can be solved. Furthermore, it requires some boilerplate code such as
functor instances for the data types. In return, pattern matching is easier and does
not require dynamic typing. Classy deep embedding only strains the programmer
with writing the extension case for the main data type and the occasional loopback
constructor.

Löh and Hinze (2006) proposed a language extension that allows open data
types and open functions, i.e. functions and data types that can be extended with
more cases later on. They hinted at the possibility of using type classes for open
functions but had serious concerns that pattern matching would be crippled because
constructors are becoming types, thus ultimately becoming impossible to type.
In contrast, this chapter shows that pattern matching is easily attainable—albeit
using dynamic types—and that the terms can be typed without complicated type
system extensions.

A technique similar to classy deep embedding was proposed by Najd and Peyton
Jones (2017) to tackle a slightly different problem, namely that of reusing a data
type for multiple purposes in a slightly different form. For example to decorate the
abstract syntax tree of a compiler differently for each phase of the compiler. They
propose to add an extension descriptor as a type variable to a data type and a type
family that can be used to decorate constructors with extra information and add
additional constructors to the data type using an extension constructor. Classy
deep embedding works similarly but uses existentially quantified type variables to
describe possible extensions instead of type variables and type families. In classy
deep embedding, the extensions do not need to be encoded in the type system and
less boilerplate is required. Furthermore, pattern matching on extensions becomes

a bit more complicated but in return it allows for multiple extensions to be added orthogonally and avoids the necessity for type system extensions.

Tagless-final embedding is the shallowly embedded counterpart of classy deep embedding and was invented for the same purpose; overcoming the issues with standard shallow embedding (Carette et al., 2009). Classy deep embedding was organically grown from observing the evolution of tagless-final embedding. The main difference between tagless-final embedding and classy deep embedding—and in general between shallow and deep embedding—is that intensional analyses of the abstract syntax tree are more difficult because there is no tangible abstract syntax tree data structure. In classy deep embedding, it is possible to define transformations even across extensions. Furthermore, in classy deep embedding, defining (mutual) dependent interpretations is automatically supported whereas in tagless-final embedding this requires some amount of code duplication (Sun et al., 2022).

Hybrid approaches between deep and shallow embedding exist as well. For example, Svenningsson and Axelsson (2013) show that by expressing the deeply embedded language in a shallowly embedded core language, extensions can be made orthogonally as well. Classy deep embedding differs from the hybrid approaches in the sense that it does not require the language extensions to be expressible in the core language.

2.8.1   Comparison

No single DSL embedding technique is the silver bullet, there is no way of perfectly satisfying all requirements programmers have. Sun et al. (2022) provided a thorough comparison of embedding techniques including more axes than just the two stated in the expression problem.

Table 2.1 shows a variant of their comparison table. The first two rows describe the two axes of the original expression problem and the third row describes the added axis of modular dependency handling as stated by Sun et al. The *poly.* style of embedding—including tagless-final—falls short of this requirement.

Intensional analysis is an umbrella term for pattern matching and transformations. In shallow embedding, intensional analysis is more complex and requires stateful views describing context, but it is possible to implement though.

Simple type system describes whether it is possible to encode this embedding technique without many type system extensions. In classy deep embedding, there is either a bit more scaffolding and boilerplate required or advanced type system extensions need to be used.

Minimal boilerplate denotes the amount of scaffolding and boilerplate required. For example, hybrid embedding requires a transcoding step between the deep syntax and the shallow core language.

Table 2.1: Comparison of embedding techniques, extended from Sun et al. (2022, section 3.6).

	Shallow	Deep	Hybrid	Poly.	Comp.	à la	Classy
Extend constructs	●	○	◐[1]	●	●	●	●
Extend views	○	●	●	●	●	●	●
Modular dependencies	○	●	●	○	●	●	●
Intensional analysis	◐[2]	●	●	◐[2]	◐[2]	●	●[3]
Simple type system	●	●	○	●	●	○	●[4]
Minimal boilerplate	●	●	○	●	●	○	●[4]

[1] Only if the extension is expressible in the core language.
[2] Requires ingenuity and are sometimes awkward to write.
[3] Cross-extensional pattern matching requires *safe* dynamic typing.
[4] Either a simple type system or little boilerplate.

2.9   Conclusion

Classy deep embedding is a novel organically grown embedding technique that alleviates deep embedding from the extensibility problem in most cases.

By abstracting the semantics functions to type classes they become overloaded in the language constructs. This upgrade makes it possible to add new language constructs in a separate type. These extensions are brought together in a special extension constructor residing in the main data type. This extension case is overloaded by the language construct using a data type containing the class dictionary. As a result, orthogonal extension is possible for language constructs and semantics using only little syntactic overhead or type annotations. The basic technique only requires—well established through history and relatively standard—existential data types. However, if needed, the technique generalises to GADTs as well, adding rank-2 types to the list of type system requirements as well. Finally, the abstract syntax tree remains observable which makes it suitable for intensional analyses, albeit using occasional dynamic typing for truly cross-extensional transformations.

Defining reuseable expressions overloaded in semantics or using multiple semantics on a single expression requires some boilerplate still, getting around these issues remains future work. Section 2.A shows how the boilerplate can be minimised using advanced type system extensions.

Acknowledgements

2.A Reprise: reducing boilerplate

One of the unique selling points of classy deep embedding is that it, in its basic form, does not require advanced type system extensions nor a lot of boilerplate. However, generalising the technique to GADTs arguably unleashes a cesspool of *unsafe* compiler extensions. If we are willing to work with extensions, almost all the boilerplate can be inferred or generated.

In classy deep embedding, the DSL datatype is parametrised by a type variable providing a witness to the interpretation on the language. When using multiple interpretations, these need to be bundled in a data type. Using the GHC's `ConstraintKind` extension, we can make these witnesses explicit, tying into Haskell's type system immediately. Furthermore, this constraint does not necessarily have to be a single constraint, after enabling `DataKinds` and `TypeOperators`, we can encode lists of witnesses instead (Yorgey et al., 2012). The data type for this list of witnesses is `Record` as shown in listing 2.1. This GADT is parametrised by two type variables. The first type variable (`dt`) is the type or type constructor on which the constraints can be applied and the second type variable (`clist`) is the list of constraints constructors itself. This means that when `Cons` is pattern matched, the overloading of the type class constraint for `c dt` can be solved by the compiler. `KindSignatures` is used to force the kinds of the type parameters and the kind of `dt` is polymorphic (`PolyKinds`) so that the `Record` data type can be used for DSLs using type classes but also type constructor classes (e.g. when using GADTs).

```haskell
data Record (dt :: k) (clist :: [k → Constraint]) where
    Nil  :: Record dt '[]
    Cons :: c dt ⇒ Record dt cs → Record dt (c ': cs)
```
Listing (Haskell) 2.1: Data type for a list of constraints.

To incorporate this type in the `Expr` type, the `Ext` constructor changes from containing a single witness dictionary to a `Record` type containing all the required dictionaries.

```haskell
data Expr c
    = Lit Int
    | Add (Expr c) (Expr c)
    | Ext (Record x c) x
```
Listing (Haskell) 2.2: Main data type with extension constructor.

Furthermore, we define a type class (`In`) that allows us to extract explicit dictionaries `Dict` from these records if the constraint can is present in the list. Since the constraints become available as soon as the `Cons` constructor is matched, the implementation is a type-level list traversal.

```haskell
class c `In` cs where
    project :: Record dt cs → Dict (c dt)
instance {-# OVERLAPPING #-} c `In` (c ': cs) where
    project (Cons _) = Dict
instance {-# OVERLAPPING #-} c `In` cs ⇒ c `In` (b ': cs) where
```

```haskell
    project (Cons xs) = project xs
```
Listing (Haskell) 2.3: Membership functions for constraints.

The final scaffolding is a multi-parameter type class `CreateRecord` (requiring the `MultiParamTypeclasses` and `FlexibleInstances` extension) to create these `Record` witnesses automatically. This type class creates a record structure cons by cons if and only if all type class constraints are available in the list of constraints. It is not required to provide instances for this for specific records or type classes, the two instances describe all the required constraints.

```haskell
class CreateRecord dt c where
    createRecord :: Record dt c
instance CreateRecord d '[] where
    createRecord = Nil
instance (c (d c0), CreateRecord (d c0) cs) ⇒
        CreateRecord (d c0) (c ': cs) where
    createRecord = Cons createRecord
```
Listing (Haskell) 2.4: Functions for creating the witness records.

The class constraints for the interpretation instances can now be greatly simplified, as shown in the evaluation instance for `Expr`. The implementation remains the same, only that for the extension case, a trick needs to be applied to convince the compiler of the correct instances. Using `` `In` ``'s `project` function, a dictionary can be brought into scope. This dictionary can then subsequently be used to apply the type class function on the extension using the `withDict` function from the `Data.Constraint` library.[6] The `ScopedTypeVariables` extension is used to make sure the existentially quantified type variable for the extension is matched to the type of the dictionary. Furthermore, because the class constraint is not smaller than the instance head, `UndecidableInstances` should be enabled.

```haskell
class Eval v where
    eval :: v → Int

instance Eval `In` s ⇒ Eval (Expr s) where
    eval (Lit i) = i
    eval (Add l r) = eval l + eval r
    eval (Ext r (e :: x)) = withDict (project r :: Dict (Eval x)) eval e
```

Smart constructors need to be adapted as well, as can be seen from the smart constructor `subst`. Instead of a `GDict` class constraint, a `CreateRecord` class constraint needs to be added.

```haskell
subst :: (Typeable c, CreateRecord (Subt c) c)
    ⇒ Expr c → Expr c → Expr c
subst l r = Ext createRecord (l `Subt` r)
```

Finally, defining terms in the language can be done immediately if the interpretations are known. For example, if we want to print and/or optimise the term $(\,(42 + (38 - 4)))$, we can define it as follows:

---

[6]`withDict :: Dict c → (c ⇒ r) → r`

```
e0 :: Expr '[Print,Opt]
e0 = neg (neg (Lit 42 `Add` (Lit 38 `subt` Lit 4)))
```

It is also possible to define terms in the DSL as being overloaded in the interpretation. This does require enumerating all the `CreateRecord` type classes for every extension similarly as was required for `GDict`. At the call site, the concrete list of constraints must be known.

```
e₁ :: (Typeable c, CreateRecord (Neg c) c, CreateRecord (Subst c) c)
   ⇒ Expr c
e₁ = neg (neg (Lit 42 `Add` (Lit 38 `subt` Lit 4)))
```

Finally, using the `TypeFamilies` extension, type families can be created for bundling `In` constraints (`UsingExt`) and `CreateRecord` constraints (`DependsOn`), making the syntax even more descriptive. E.g. `UsingExt '[A, B, C] c` expands to `(CreateRecord (A c) c, CreateRecord (B c) c, CreateRecord (C c) c)`. Similarly, `DependsOn '[A, B, C] s` expands to `(A `In` s, B `In` s, C `In` s)`.

```
type family UsingExt cs c :: Constraint where
    UsingExt '[] c = ()
    UsingExt (d ': cs) c = (CreateRecord (d c) c, UsingExt cs c)

type family DependsOn cs c :: Constraint where
    DependsOn '[] c = ()
    DependsOn (d ': cs) c = (d `In` c, DependsOn cs c)
```

Defining the previous expression can now be done with the following shortened type that describes the semantics better:

```
e₁ :: (Typeable c, UsingExt '[Neg, Subst]) ⇒ Expr c
```

Giving an instance for `Interp` for `DataType` that uses the extensions `e_1`, `e₂`, $\cdots$ and depends on interpretations `i_1,i_2`, $\cdots$ is done as follows:

```
instance ( UsingExt  '[e_1, e_2, ⋯] s, DependsOn '[i_1, i_2, ⋯] s)
       ⇒ Interp (DataType s) where
   ...
```

With these enhancements, there is hardly any boilerplate required to use classy deep embedding. The `Record` data type; the `CreateRecord` type class; and the `UsingExt` and `DependsOn` type families can be provided as a library only requiring the programmer to create the extension constructors with their respective implementations and smart constructors for language construct extensions. The source code for this extension can be found here: https://gitlab.com/mlubbers/classydeepembedding.[7] It contains examples for expressions, expressions using GADTs, detection of sharing in expressions (modelled after Kiselyov (2011)), a GADT version of sharing detection, and a region DSL (modelled after Sun et al. (2022)).

---

[7]Lubbers, M. (2022): Library and examples for enhanced classy deep embedding. Zenodo. 10.5281/zenodo.7277498.

2.B   Data types and definitions

This appendix contains all definitions omitted for brevity.

```
data Sub_g d a where
    Sub_g       :: (Eq a, Num a)        ⇒ Expr_g d a → Expr_g d a → Sub_g d a
    SubLoop_g ::                           Expr_g d a → Sub_g d a

data Eq_g d a  where
    Eq_g        :: (Typeable a, Eq a) ⇒ Expr_g d a → Expr_g d a → Eq_g d Bool
    EqLoop_g  ::                          Expr_g d a → Eq_g d a
```
Listing (Haskell) 2.5: Data type definitions.

```
sub_g :: (Typeable d, GDict (d (Sub_g d)), Eq a, Num a) ⇒
    Expr_g d a → Expr_g d a → Expr_g d a
sub_g e₁ e₂ = Ext_g gdict (Sub_g e₁ e₂)

eq_g :: (Typeable d, GDict (d (Eq_g d)), Eq a, Typeable a) ⇒
    Expr_g d a → Expr_g d a → Expr_g d Bool
eq_g e₁ e₂ = Ext_g gdict (Eq_g e₁ e₂)
```
Listing (Haskell) 2.6: Smart constructors.

```
newtype PrintDict_g v = PrintDict_g (forall a.v a → String)

class HasPrint_g d where
    getPrint_g :: d v → v a → String

instance HasPrint_g PrintDict_g where
    getPrint_g (PrintDict_g e) = e

newtype OptDict_g v = OptDict_g (forall a.v a → v a)

class HasOpt_g d where
    getOpt_g :: d v → v a → v a

instance HasOpt_g OptDict_g where
    getOpt_g (OptDict_g e) = e
```
Listing (Haskell) 2.7: Semantics classes and data types.

```
instance Print_g v ⇒ GDict (PrintDict_g v) where
    gdict = PrintDict_g print_g
instance Opt_g v   ⇒ GDict (OptDict_g v)   where
    gdict = OptDict_g opt_g
```
Listing (Haskell) 2.8: `GDict` instances.

```
instance HasEval_g d ⇒ Eval_g (Expr_g d) where
    eval_g (Lit_g v)     = v
    eval_g (Add_g e₁ e₂) = eval_g e₁ + eval_g e₂
    eval_g (Ext_g d x)   = getEval_g d x

instance HasEval_g d ⇒ Eval_g (Sub_g d) where
    eval_g (Sub_g      e₁ e₂) = eval_g e₁ - eval_g e₂
    eval_g (SubLoop_g e)      = eval_g e
```

```haskell
instance HasEvalg d ⇒ Evalg (Negg d) where
    evalg (Negg     e) = negate (evalg e)
    evalg (NegLoopg e) = evalg e

instance HasEvalg d ⇒ Evalg (Eqg d) where
    evalg (Eqg      e1 e2) = evalg e1 == evalg e2
    evalg (EqLoopg e)      = evalg e

instance HasEvalg d ⇒ Evalg (Notg d) where
    evalg (Notg     e) = not (evalg e)
    evalg (NotLoopg e) = evalg e
```

Listing (Haskell) 2.9: Evaluator instances.

```haskell
instance HasPrintg d ⇒ Printg (Exprg d) where
    printg (Litg v)     = show v
    printg (Addg e1 e2) = "(" ++ printg e1 ++ "+" ++ printg e2 ++ ")"
    printg (Extg d x)   = getPrintg d x

instance HasPrintg d ⇒ Printg (Subg d) where
    printg (Subg     e1 e2) = "(" ++ printg e1 ++ "−" ++ printg e2 ++ ")"
    printg (SubLoopg e)     = printg e

instance HasPrintg d ⇒ Printg (Negg d) where
    printg (Negg     e) = "(negate " ++ printg e ++ ")"
    printg (NegLoopg e) = printg e

instance HasPrintg d ⇒ Printg (Eqg d) where
    printg (Eqg      e1 e2) = "(" ++ printg e1 ++ "==" ++ printg e2 ++ ")"
    printg (EqLoopg e)      = printg e

instance HasPrintg d ⇒ Printg (Notg d) where
    printg (Notg     e) = "(not " ++ printg e ++ ")"
    printg (NotLoopg e) = printg e
```

Listing (Haskell) 2.10: Printer instances.

```haskell
instance HasOptg d ⇒ Optg (Exprg d) where
    optg (Litg v)     = Litg v
    optg (Addg e1 e2) = case (optg e1, optg e2) of
        (Litg 0, e2'  ) → e2'
        (e1',    Litg 0) → e1'
        (e1',    e2'  ) → Addg e1' e2'
    optg (Extg d x)   = Extg d (getOptg d x)

instance HasOptg d ⇒ Optg (Subg d) where
    optg (Subg     e1 e2) = case (optg e1, optg e2) of
        (e1', Litg 0) → SubLoopg e1'
        (e1', e2'  ) → Subg e1' e2'
    optg (SubLoopg e) = SubLoopg (optg e)

instance (Typeable d, GDict (d (Negg d)), HasOptg d) ⇒ Optg (Negg d) where
    optg (Negg (Addg e1 e2))
        = NegLoopg (Addg (optg (negg e1)) (optg (negg e2)))
    optg (Negg (Extg d x))
        = case fromDynamic (toDyn (getOptg d x)) of
            Just (Negg e) → NegLoopg e
```

```
                 _              → Negg (Extg d (getOptg d x))
    optg (Negg     e) = Negg (optg e)
    optg (NegLoopg e) = NegLoopg (optg e)

instance HasOptg d ⇒ Optg (Eqg d) where
    optg (Eqg      e₁ e₂) = Eqg (optg e₁) (optg e₂)
    optg (EqLoopg e)      = EqLoopg (optg e)
```

Listing (Haskell) 2.11: Optimisation instances.

Chapter 3

First-class data types in shallow embedded domain-specific languages using metaprogramming

FP languages are excellent for hosting eDSLs because of their rich type systems, minimal syntax, and referential transparency. However, data types defined in the host language are not automatically available in the embedded language. To do so, all the operations on the data type must be ported to the eDSL resulting in a lot of boilerplate.

This chapter shows that by using metaprogramming, all first-order user-defined data types can be automatically made first class in shallow eDSLs. We show this by providing an implementation in Template Haskell (TH) for a typical DSL with two different semantics. Furthermore, we show that by utilising quasiquotation, there is hardly any burden on the syntax. Finally, the chapter also serves as a gentle introduction to TH.

3.1 Introduction

FP languages are excellent candidates for hosting eDSLs because of their rich type systems, minimal syntax, and referential transparency. By expressing the language constructs in the host language, the parser, the type checker, and the run time can be inherited from the host language. Unfortunately, data types defined in the host language are not automatically available in the eDSL.

The two main strategies for embedding DSLs in an FP language are deep embedding (also called initial) and shallow embedding (also called final). Deep embedding represents the constructs in the language as data types and the semantics as functions over these data types. This makes extending the language with new semantics effortless: just add another function. In contrast, adding language constructs requires changing the data type and updating all existing semantics to support this new construct. Shallow embedding on the other hand models the language constructs as functions with the semantics embedded. Consequently, adding a construct is easy, i.e. it only entails adding another function. Contrarily, adding semantics requires adapting all language constructs. Lifting the functions to type classes, i.e. parametrising the constructs over the semantics, allows extension of the language both in constructs and in semantics orthogonally. This advanced style of embedding is called tagless-final or class-based shallow embedding (Kiselyov, 2012).

While it is often possible to lift values of a user-defined data type to a value in the DSL, it is not possible to interact with it using DSL constructs, since they are not first-class citizens.

Concretely, it is not possible to 1. construct values from expressions using a constructor, 2. deconstruct values into expressions using a deconstructor or pattern matching, 3. test which constructor the value holds. The functions for this are simply not available automatically in the embedded language. For some semantics— such as an interpreter—it is possible to directly lift the functions from the host language to the DSL. In other cases—e.g. *compiling* DSLs such as a compiler or a printer—this is not possible (Elliott et al., 2003). Thus, all the operations on the data type have to be defined by hand requiring a lot of plumbing and resulting in a lot of boilerplate code.

To relieve the burden of adding all these functions, metaprogramming—and custom quasiquoters—can be used. Metaprogramming entails that some parts of the program are generated by a program itself, i.e. the program is data. Quasiquotation is a metaprogramming mechanism that allows entering verbatim code for which a—possibly user defined—translation is used to convert the verbatim code to host language abstract syntax tree (AST) nodes. Metaprogramming allows functions to be added to the program at compile time based on the structure of user-defined data types.

3.1.1 Contributions

This chapter shows that with the use of metaprogramming, all first-order user-defined data types can automatically be made first class for shallow eDSLs. It does so by providing an implementation in TH for a typical DSL with two different semantics: an interpreter and a pretty printer. Furthermore, we show that by utilising quasiquotation, there is hardly any burden on the syntax. Finally, the chapter also serves as a gentle introduction to TH and reflects on the process of using TH.

3.2   Tagless-final embedding

Tagless-final embedding is an upgrade to standard shallow embedding achieved by lifting all language construct functions to type classes. As a result, views on the DSL are data types implementing these classes.

To illustrate the technique, a simple DSL, a language consisting of literals and addition, is outlined. This language, implemented according to the tagless-final style (Carette et al., 2009) in Haskell (Peyton Jones, 2003) consists initially only of one type class containing two functions. The `lit` function lifts values from the host language to the DSL domain. The class constraint `Show` is enforced on the type variable `a` to make sure that the value can be printed. The infix function `+.` represents the addition of two expressions in the DSL.

```
class Expr v where
    lit :: Show a ⇒ a → v a
    (+.) :: Num a ⇒ v a → v a → v a
infixl 6 +.
```

The implementation of a view on the DSL is achieved by implementing the type classes with the data type representing the view. In the case of our example DSL, an interpreter accounting for failure may be implemented as an instance for the `Maybe` type. The standard infix functor application and infix sequential application are used so that potential failure is abstracted away from.[1]

```
instance Expr Maybe where
    lit a = Just a
    (+.) l r = (+) <$> l <*> r
```

3.2.1   Adding language constructs

To add an extra language construct we define a new class housing it. For example, to add division we define a new class as follows:

```
class Div v where
    (/.) :: Integral a ⇒ v a → v a → v a
infixl 7 /.
```

Division is an operation that is undefined if the right operand is equal to zero. To capture this behaviour, the `Nothing` constructor from `Maybe` is used to represent errors. Both sides of the division operator are evaluated. If the right-hand side is zero, the division is not performed and an error is returned instead:

```
instance Div Maybe where
    (/.) l r = l >>= \x→r >>= \y→
        if y == 0 then Nothing else Just (x `div` y)
```

```
    <$> ::   (a → b) → f a → f b
1   <*> :: f (a → b) → f a → f b
    infixl 4 <$>, <*>
```

3.2.2   Adding semantics

To add semantics to the DSL, the existing classes are implemented with a novel
data type representing the view on the DSL. First a data type representing the
semantics is defined. In this case, the printer is kept very simple for brevity and
just defined as a **newtype** of a string to store the printed representation.[2]   Since
the language is typed, the printer data type has to have a type variable, but it is
only used during typing—i.e. a phantom type (Leijen and Meijer, 2000):

```
newtype Printer a = P { runPrinter :: String }
```

The class instances for `Expr` and `Div` for the pretty printer are straightforward
and as follows:

```
instance Expr Printer where
    lit a = P (show a)
    (+.) l r = P ("(" ++ runPrinter l
             ++ "+" ++ runPrinter r ++ ")")

instance Div Printer where
    (/.) l r = P ("(" ++ runPrinter l
             ++ "/" ++ runPrinter r ++ ")")
```

3.2.3   Functions

Adding functions to the language is achieved by adding a multi-parameter class
to the DSL. The type of the class function allows for the implementation to only
allow first-order functions by supplying the arguments in a tuple. Furthermore,
with the `:-` operator the syntax becomes useable. Finally, by defining the functions
as a high-order abstract syntax (HOAS) type safety is achieved (Chlipala, 2008;
Pfenning and Elliott, 1988). The complete definition looks as follows:

```
class Function a v where
    fun :: ((a → v s) → In (a → v s) (v u)) → v u
data In a b = a :- b
infix 1 :-
```

The `Function` type class is now used to define functions with little syntactic
overhead.[3]   The following listing shows an expression in the DSL utilising two
user-defined functions:

```
    fun \increment→ (\x     →x +. lit 1)
:- fun \divide→    (\(x, y)→x /. y    )
:- increment (divide (lit 38, lit 5))
```

---

[2]In this case a **newtype** is used instead of regular **data** declarations. **newtype**s are special
data types only consisting a single constructor with one field to which the type is isomorphic.
During compilation the constructor is completely removed resulting in no overhead (Peyton Jones,
2003, section 4.2.3).

[3]The `BlockArguments` extension of GHC is used to reduce the number of brackets that allows
lambda's to be an argument to a function without brackets

The interpreter only requires one instance of the `Function` class that works for any argument type. In the implementation, the resulting function `g` is simultaneously provided to the definition `def`. Because the laziness of Haskell's lazy let bindings, this results in a fixed point calculation:

```
instance Function a Maybe where
    fun def = let g :- m = def g in m
```

The given `Printer` type is not sufficient to implement the instances for the `Function` class, it must be possible to generate fresh function names. After extending the `Printer` type to contain some sort of state to generate fresh function names and a `MonadWriter [String]`[4] to streamline the output, we define an instance for every arity.   To illustrate this, the instance for unary functions is shown, all other arities are implemented in similar fashion.

```
instance Function () Printer where ···
instance Function (Printer a) Printer where ···
    fun def = freshLabel >>= \f →
        let g :- m = def $ \a0 → const undefined
                <$> (tell ["f", show f, " ("]
                    >> a0 >> tell [")"])
        in  tell ["let f", f, " a0 = "]
                >> g (const undefined <$> tell ["a0"])
        >>  tell [" in "] >> m
instance Function (Printer a, Printer b) Printer where ···
```

Running the given printer on the example code shown before produces roughly the following output, running the interpreter on this code results in `Just 8`.

```
    let f0 a1 = a1 + 1
in let f2 a3 a4 = a3 / a4
in f0 (f2 38 5)
```

3.2.4   Data types

Lifting values from the host language to the DSL is possible using the `lit` function as long as the type of the value has instances for all the class constraints. Unfortunately, once lifted, it is not possible to do anything with values of the user-defined data type other than passing them around. It is not possible to construct new values from expressions in the DSL, to deconstruct a value into the fields, nor to test of which constructor the value is. Furthermore, while in our language the only constraint is the automatically derivable `Show`, in real-world languages the class constraints may be very difficult to satisfy for complex types, for example serialisation to a single stack cell in the case of a compiler.

As a consequence, for user-defined data types—such as a programmer-defined list type[5]—to become first-class citizens in the DSL, language constructs for constructors, deconstructors and constructor predicates must be defined. Field

---

[4] `freshLabel :: Printer String`
`tell :: MonadWriter w m ⇒ w → m ()`

selectors are also useful functions for working with user-defined data types. They are not considered for the sake of brevity but can be implemented using the deconstructor functions. The constructs for the list type would result in the following class definition:

```
class ListDSL v where
    — constructors
    nil    :: v (List a)
    cons   :: v a → v (List a) → v (List a)
    — deconstructors
    unNil  :: v (List a) → v b → v b
    unCons :: v (List a) → (v a → v (List a) → v b) → v b
    — constructor predicates
    isNil  :: v (List a) → v Bool
    isCons :: v (List a) → v Bool
```

Furthermore, instances for the DSL's views need to be created. For example, to use the interpreter, the following instance must be available. Note that at first glance, it would feel natural to have `isNil` and `isCons` return `Nothing` since we are in the `Maybe` monad. However, this would fail the entire expression and the idea is that the constructor test can be done from within the DSL.

```
instance ListDSL Maybe where
    nil        = Just Nil
    cons hd tl = Cons <$> hd <*> tl
    unNil  d f = d >>= \Nil→f
    unCons d f = d >>= \(Cons hd tl)→f (Just hd) (Just tl)
    isNil  d   = d >>= \case⁶
        Nil → Just True
        _   → Just False
    isCons d   = d >>= \case
        Cons _ _ → Just True
        Nil      → Just False
```

Adding these classes and their corresponding instances is tedious and results in boilerplate code. We therefore resort to metaprogramming, and in particular TH (Sheard and Peyton Jones, 2002) to alleviate this burden.

3.3 Template metaprogramming

Metaprogramming is a special flavour of programming where programs have the ability to treat and manipulate programs or program fragments as data. There are several techniques to facilitate metaprogramming, moreover it has been around for many years now (Lilis and Savidis, 2019). Even though it has been around for many years, it is considered complex (Sheard, 2001).

TH is GHC's de facto metaprogramming system, implemented as a compiler extension together with a library (Sheard and Peyton Jones, 2002)(GHC Team,

---

[5]For example: **data List a = Nil | Cons {hd :: a, tl :: List a}**

[6]**\case** is an abbreviation for **\x→case x of** ⋯ when using GHC's `LambdaCase` extension.

2021b, section 6.13.1). Readers already familiar with TH can safely skip this section.

TH adds four main concepts to the language, namely AST data types, splicing, quasiquotation and reification. With this machinery, regular Haskell functions can be defined that are called at compile time, inserting generated code into the AST. These functions are monadic functions operating in the **Q** monad. The **Q** monad facilitates failure, reification and fresh identifier generation for hygienic macros (Kohlbecker et al., 1986). Within the **Q** monad, capturable and non-capturable identifiers can be generated using the mkName and newName functions respectively. The *Peter Parker principle*[7] holds for the **Q** monad as well because it executes at compile time and is very powerful. For example, it can subvert module boundaries, thus accessing constructors that were hidden; access the structure of abstract types; and it may cause side effects during compilation because it is possible to call IO operations (Terei et al., 2012). To achieve the goal of embedding data types in a DSL we refrain from using these *unsafe* features.

3.3.1 Data types

For all of Haskell's AST elements, data types are provided that are mostly isomorphic to the actual data types used in the compiler. With these data types, the entire syntax of a Haskell program can be specified. Often, a data type is suffixed with the context, e.g. there is a VarE and a VarP for a variable in an expression or in a pattern respectively. To give an impression of these data types, a selection of data types available in TH is given below:

```
data Dec = FunD Name [Clause] | DataD Cxt Name ··· | SigD Name Type
    | ClassD Cxt Name | ···
data Clause = Clause [Pat] Body [Dec]
data Pat = LitP Lit | VarP Name | TupP [Pat] | WildP | ListP [Pat] | ···
data Body = GuardedB [(Guard, Exp)] | NormalB Exp
data Guard = NormalG Exp | PatG [Stmt]
data Exp = VarE Name | LitE Lit | AppE Exp Exp | TupE [Maybe Exp]
    | LamE [Pat] Exp | ···
data Lit = CharL Char | StringL String | IntegerL Integer | ···
```

To ease creating AST data types in the **Q** monad, lowercase variants of the constructors are available that lift the constructor to the **Q** monad. For example, for the LamE constructor, the following lamE function is available.

```
lamE :: [Q Pat] → Q Exp → Q Exp
lamE ps es = LamE <$> sequence ps <*> es
```

3.3.2 Splicing

Special splicing syntax ($(· · ·)) marks functions for compile-time execution. Apart from the fact that they always produce a value of an AST data type, they are regular functions. Depending on the context and location of the splice, the result

---

[7]With great power comes great responsibility.

type is either a list of declarations, a type, an expression or a pattern. The result of this function, when executed successfully, is then spliced into the code and treated as regular code by the compiler. Consequently, the code that is generated may not be type safe, in which case the compiler provides a type error on the generated code. The following listing shows an example of a TH function generating on-the-fly functions for arbitrary field selection in a tuple. When called as `$(tsel 2 4)` it expands at compile time to `\(_, _, f, _)→f`:

```
tsel :: Int → Int → Q Exp
tsel field total = do
    f ← newName "f"
    lamE [ tupP [if i == field then varP f else wildP
         | i←[0..total-1]]] (varE f)
```

3.3.3 Quasiquotation

Another key concept of TH is Quasiquotation, the dual of splicing (Bawden, 1999). While it is possible to construct entire programs using the provided data types, it is a little cumbersome. Using *Oxford brackets* (`[| ... |]`) or single or double apostrophes, verbatim Haskell code can be entered which is converted automatically to the corresponding AST nodes easing the creation of language constructs. Depending on the context, different quasiquotes are used: • $[\![\cdots]\!]$ or $[\![_e \cdots]\!]$ for expressions • $[\![_d \cdots]\!]$ for declarations • $[\![_p \cdots]\!]$ for patterns • $[\![_t \cdots]\!]$ for types • `'`$\cdots$ for function names • `''`$\cdots$ for type names. It is possible to escape the quasiquotes again by splicing. Variables defined within quasiquotes are always fresh—as if defined with `newName`—but it is possible to capture identifiers using `mkName`. For example, $[\![\x→x]\!]$ translates to `newName "x" >>= \x→lamE [varP x] (varE x)` and does not interfere with other `x`s already defined.

3.3.4 Reification

Reification is the act of querying the compiler for information about a certain name. For example, reifying a type name results in information about the type and the corresponding AST nodes of the type's definition. This information can then be used to generate code according to the structure of data types. Reification is done using the `reify :: Name → Q Info` function. The `Info` type is an algebraic data type (ADT) containing all the—known to the compiler—information about the matching type: constructors, instances, *&c.*

3.4 Metaprogramming for generating DSL functions

With the power of metaprogramming, we can generate the boilerplate code for our user-defined data types automatically at compile time. To generate the code required for the DSL, we define the `genDSL` function. The type belonging to the name passed as an argument to this function is made available for the DSL by

generating the `typeDSL` class and view instances. For the `List` type it is called as:
`$(genDSL ''List)`.[8]

The `genDSL` function is a regular function—though TH requires that it is defined in a separate module—that has type: `Name → Q [Dec]`, i.e. given a name, it produces a list of declarations in the **Q** monad. The `genDSL` function first reifies the name to retrieve the structural information. If the name matches a type constructor containing a data type declaration, the structure of the type—the type variables, the type name and information about the constructors[9]—are passed to the `genDSL'` function. The `getConsName` function filters out unsupported data types such as GADTs and makes sure that every field has a name. For regular ADTs, the `adtFieldName` function is used to generate a name for the constructor based on the indices of the fields.[10]   From this structure of the type, `genDSL'` generates a list of declarations containing a class definition (section 3.4.1), instances for the interpreter (section 3.4.2), and instances of the printer (section 3.4.3) respectively.

```
genDSL :: Name → Q [Dec]
genDSL name = reify name >>= \case
    TyConI (DataD cxt typeName tvs mkind constructors derives)
        → mapM getConsName constructors >>= \d→genDSL' tvs typeName d
    t → fail ("genDSL does not support: " ++ show t)

getConsName :: Con → Q (Name, [VarBangType])
getConsName (NormalC consName fs) = pure (consName,
    [(adtFieldName consName i, b, t) | (i, (b, t))←[0..] `zip` fs])
getConsName (RecC consName fs) = pure (consName, fs)
getConsName c = fail ("genDSL does not support: " ++ show c)

genDSL' :: [TyVarBndr] → Name → [(Name, [VarBangType])] → Q [Dec]
genDSL' typeVars typeName constructors = sequence
    [ mkClass, mkInterpreter, mkPrinter, ··· ]
  where
    (consNames, fields) = unzip constructors
    ...
```

3.4.1   Class generation

The function for generating the class definition is defined in the **where** clause of the `genDSL'` function. Using the `classD` constructor, a single type class is created with a single type variable `v`. The `classD` function takes five arguments: 1. a context, i.e. the class constraints, which is empty in this case 2. a name, generated from the type name using the `className` function that simply appends the text `DSL` 3. a list of type variables, in this case the only type variable is the view on the DSL, i.e. `v` 4. functional dependencies, empty in our case 5. a list of function declarations, i.e. the class members, in this case it is a concatenation of the constructors,

---

[8]`''` is used instead of `'` to instruct the compiler to look up the information for `List` as a type and not as a constructor.

[9]Defined as **type** `VarBangType = (Name, Bang, Type)` by TH.

[10]`adtFieldName :: Name → Integer → Name`

deconstructors, and constructor predicates Depending on the required information, either `zipWith` or `map` is used to apply the generation function to all constructors.

```
mkClass :: Q Dec
mkClass = classD (cxt []) (className typeName) [PlainTV (mkName "v")] []
    (  zipWith mkConstructor   consNames fields
    ++ zipWith mkDeconstructor consNames fields
    ++ map      mkPredicate     consNames
    )
```

In all class members, the view `v` plays a crucial role. Therefore, a definition for `v` is accessible for all generation functions. Furthermore, the `res` type represents the *result* type, it is defined as the type including all type variables. This result type is derived from the type name and the list of type variables. In case of the `List` type, `res` is defined as `v (List a)` and is available for as well:

```
v = varT (mkName "v")
res = v `appT` foldl appT (conT typeName)
    (map getName typeVars)
  where getName (PlainTV name)    = varT name
        getName (KindedTV name _) = varT name
```

Constructors

The constructor definitions are generated from just the constructor names and the field information. All class members are defined using the `sigD` constructor that represents a function signature. The first argument is the name of the constructor function, a lowercase variant of the actual constructor name generated using the `constructorName` function. The second argument is the type of the function. A constructor C_k of type T where $T\ tv_0\ \dots\ tv_n = \dots \mid C_k\ a_0\ \dots\ a_m \mid \dots$ is defined as a DSL function $c_k :: v\ a_0 \to \dots \to v\ a_m \to v\ (T\ v_0\ \dots\ v_n)$. In the implementation, first the view `v` is applied to all the field types. Then, the constructor type is constructed by folding over the lifted field types with the result type as the initial value using `mkCFun`.

```
mkConstructor :: Name → [VarBangType] → Q Dec
mkConstructor n fs = sigD (constructorName n) (mkCFun fs res)

mkCFun :: [VarBangType] → Q Type → Q Type
mkCFun fs res = foldr (\x y→⟦ₜ $x  →  $y⟧)
    (map (\(_, _, t)→v `appT` pure t) fs)
```

Deconstructors

The deconstructor is generated similarly to the constructor as the function for generating the constructor is the second argument modulo change in the result type. A deconstructor C_k of type T is defined as a DSL function $unC_k :: v\ (T\ v_0 \dots v_n) \to (v\ a_0 \to \dots \to v\ a_m \to v\ b) \to v\ b$. In the implementation, `mkCFun` is reused to construct the type of the deconstructor as follows:

```
mkDeconstructor :: Name → [VarBangType] → Q Dec
mkDeconstructor n fs = sigD (deconstructorName n)
    ⟦ₜ $res → $(mkCFun fs ⟦ₜ $v $b⟧) → $v $b⟧
  where b = varT (mkName "b")
```

Constructor predicates

The last part of the class definition consists of the constructor predicates, a function that checks whether the provided value of type T contains a value with constructor C_k. A constructor predicate for constructor C_k of type T is defined as a DSL function $isC_k :: v (T v_0 \ldots v_n) \rightarrow v Bool$. A constructor predicate—name prefixed by `is`—is generated for all constructors. They all have the same type:

```
mkPredicate :: Name → Q Dec
mkPredicate n = sigD (predicateName n) ⟦ₜ $res → $v Bool⟧
```

3.4.2   Interpreter instance generation

Generating the interpreter for the DSL means generating the class instance for the `Interpreter` data type using the `instanceD` function. The first argument of the instance is the context, this is left empty. The second argument of the instance is the type, the `Interpreter` data type applied to the class name. Finally, the class function instances are generated using the information derived from the structure of the type. The structure for generating the function instances is very similar to the class definition, only for the function instances of the constructor predicates, the field information is required as well as the constructor names.

```
mkInterpreter :: Q Dec
mkInterpreter = instanceD (cxt [])
        ⟦ₜ $(conT (className typeName)) Interpreter⟧
    (   zipWith mkConstructor consNames fields
    ++ zipWith mkDeconstructor consNames fields
    ++ zipWith mkPredicate consNames fields)
  where ···
```

Constructors

The interpreter is a view on the DSL that immediately executes all operations in the `Maybe` monad. Therefore, the constructor function can be implemented by lifting the actual constructor to the `Maybe` type using sequential application. I.e. for a constructor C_k this results in the following constructor: `ck a0 ··· am = pure Ck <*> a0 <*> ··· <*> am`. To avoid accidental shadowing, fresh names for all the arguments are generated. The `ifx` function is used as a shorthand for defining infix expressions.[11]

```
   11 ifx :: String → Q Exp → Q Exp → Q Exp
      ifx op a b = infixE (Just a) (varE (mkName op)) (Just b)
```

```
mkConstructor :: Name → [VarBangType] → Q Dec
mkConstructor consName fs = do
    fresh ← sequence [newName "a" | _←fs]
    fun (constructorName consName) (map varP fresh)
        (foldl (ifx "<*>") ⟦pure $(conE consName)⟧
        (map varE fresh))
```

Deconstructors

In the case of a deconstructor a function with two arguments is created: the object itself (`f`) and the function doing something with the individual fields (`d`). To avoid accidental shadowing first fresh names for the arguments and fields are generated. Then, a function is created with the two arguments. First `d` is evaluated and bound to a host language function that deconstructs the constructor and passes the fields to `f`. I.e. a deconstructor function C_k is defined as: `unCk d f = d >>= \(Ck a0 .. am)→f (pure a0) ··· (pure am))`.[12]

```
mkDeconstructor :: Name → [VarBangType] → Q Dec
mkDeconstructor consName fs = do
    d ← newName "d"
    f ← newName "f"
    fresh ← mapM (newName . nameBase . fst3) fs
    fun (deconstructorName consName) [varP d, varP f]
        ⟦$(varE d) >>= \($(match f))→$(fapp f fresh)⟧
  where fapp f = foldl appE (varE f) . map (\f→⟦pure $(varE f)⟧)
        match f = pure (ConP consName (map VarP f))
```

Constructor predicates

Constructor predicates evaluate the argument and make a case distinction on the result to determine the constructor. To be able to generate a valid pattern in the case distinction, the total number of fields must be known. To avoid having to explicitly generate a fresh name for the first argument, a lambda function is used. In general, the constructor selector for C_k results in the following code `isCk f = f >>= \case Ck _ ··· _ → pure True; _ → pure False`. Generating this code is done with the following function:

```
mkPredicate :: Name → [(Var, Bang, Type)] → Q Dec
mkPredicate n fs = fun (predicateName n) []
    ⟦\x→x >>= \case
        $(conP n [wildP | _←fs]) → pure True
        _                        → pure False⟧
```

---

[12]The `nameBase :: Name → String` function from the TH library is used to convert a name to a string.

3.4.3   Pretty printer instance generation

Generating the printer happen analogously to the interpreter, a class instance for
the `Printer` data type using the `instanceD` function.

```
mkPrinter :: Q Dec
mkPrinter = instanceD (cxt []) ⟦ₜ$(conT (className typeName)) Printer⟧
    (  zipWith mkConstructor consNames fields
    ++ zipWith mkDeconstructor consNames fields
    ++ map mkPredicate consNames)
```

To be able to define a printer that is somewhat more powerful, we provide
instances for `MonadWriter`; add a state for fresh variables and a context; and define
some helper functions the `Printer` datatype. The `printLit` function is a variant
of `MonadWriters` `tell` that prints a literal string, but it can be of any type (it is
a phantom type anyway). `printCons` prints a constructor name followed by an
expression, it inserts parenthesis only when required depending on the state. `paren`
always prints parenthesis around the given printer. `>->` is a variant of the sequence
operator `>>` from the `Monad` class, it prints whitespace in between the arguments.

```
printLit  :: String → Printer a
printCons :: String → Printer a → Printer a
paren     :: Printer a → Printer a
(>->)     :: Printer a1 → Printer a2 → Printer a3
pl        :: String → Q Exp
```

Constructors

For a constructor C_k the printer is defined as: `ck a0 ⋯ am = printCons "Ck"`
`(printLit "" >-> a0 >-> ⋯ >-> am)`. To generate the second argument to the
`printCons` function, a fold is used with `printLit ""` as the initial element to account
for constructors without any fields as well, e.g. `Nil` is translated to `nil = printCons`
`"Nil" (printLit "")`.

```
mkConstructor :: Name → [VarBangType]  → Q Dec
mkConstructor consName fs = do
    fresh ← sequence [newName "f" | _← fs]
    fun (constructorName consName) (map varP fresh)
        (pcons `appE` pargs fresh)
  where pcons = ⟦printCons $(lift (nameBase consName))⟧
        pargs fresh = foldl (ifx ">->") (pl "")
            (map varE fresh)
```

Deconstructors

Printing the deconstructor for C_k is defined as:

```
unCk d f
    =  printLit "unCk d"
    >-> paren (
          printLit "\(Ck" >-> printLit "a0 ⋯ am" >> printLit ")→"
```

```
      >> f (printLit "a0") ··· (printLit "am")
  )
```

The implementation for this is a little elaborate and it heavily uses the `pl` function, a helper function that translates a string literal `s` to ⟦`printLit $(lift s)`⟧, i.e. it lifts the `printLit` function to the TH domain.

```
mkDeconstructor :: Name → [VarBangType] → Q Dec
mkDeconstructor consName fs = do
    d ← newName "d"
    f ← newName "f"
    fresh ← sequence [newName "a" | _←fs]
    fun (deconstructorName consName) (map varP [d, f])
        ⟦ $(pl (nameBase (deconstructorName consName)))
        >-> $(pl (nameBase d))
        >-> paren ($(pl ('\\':'(':nameBase consName))
                   >-> $lam >> printLit ")→"
                   >>  $(hoas f))⟧
  where
    lam = pl $ unwords [nameBase f | (f, _, _)←fs]
    hoas f = foldl appE (varE f)
        [pl (nameBase f) | (f, _, _)←fs]
```

Constructor predicates

For the printer, the constructor selector for C_k results in the following code `isCk f = printLit "isCk" >-> f`.

```
mkPredicate :: Name → Q Dec
mkPredicate n = fun (predicateName n) []
    ⟦\x→ $(pl $ nameBase $ predicateName n) >-> x⟧
```

3.5   Pattern matching

It is possible to construct and deconstruct values from other DSL expressions, and to perform tests on the constructor but with a clunky and unwieldy syntax. They have become first-class citizens in a grotesque way. For example, given that we have some language constructs to denote failure and conditionals,[13] writing a list summation function in our DSL would be done as follows. For the sake of the argument we take a little shortcut here and assume that the interpretation of the DSL supports lazy evaluation by using the host language as a metaprogramming language as well, allowing us to use functions in the host language to construct expressions in the DSL.

---

[13]
```
class Support v where
    if'    :: v Bool → v a → v a → v a
    bottom :: String → v a
```

```
program :: (ListDSL v, Support v, ···) ⇒ v Int
program
    = fun \sum→(\l→if'(isNil l)
        (lit 0)
        (unCons l (\hd tl→hd +. sum tl)))
    :- sum (cons (lit 38) (cons (lit 4) nil))
```

A similar Haskell implementation is much more elegant and less cluttered because of the support for pattern matching. Pattern matching offers a convenient syntax for doing deconstruction and constructor tests at the same time.

```
sum :: List Int → Int
sum Nil = 0
sum (List hd tl) = hd + sum tl


main = sum (Cons 38 (Cons 4 Nil))
```

3.5.1 Custom quasiquoters

The syntax burden of eDSLs can be reduced using quasiquotation. In TH, quasiquotation is a convenient way to create Haskell language constructs by entering them verbatim using Oxford brackets. However, it is also possible to create so-called custom quasiquoters (Mainland, 2007). If the programmer writes down a fragment of code between tagged *Oxford brackets*, the compiler executes the associated quasiquoter functions at compile time. A quasiquoter is a value of the following data type:

```
data QuasiQuoter = QuasiQuoter
    { quoteExp  :: String → Q Exp
    , quotePat  :: String → Q Pat
    , quoteType :: String → Q Type
    , quoteDec  :: String → Q Dec
    }
```

The code between *dsl* brackets (⟦$_{dsl}$ ··· ⟧) is preprocessed by the `dsl` quasiquoter. Because the functions are executed at compile time, errors—thrown using the `MonadFail` instance of the **Q** monad—in these functions result in compile time errors. The AST nodes produced by the quasiquoter are inserted into the location and checked as if they were written by the programmer.

To illustrate writing a custom quasiquoter, we show an implementation of a quasiquoter for binary literals. The `bin` quasiquoter is only defined for expressions and parses subsequent zeros and ones as a binary number and splices it back in the code as a regular integer. Thus, ⟦$_{bin}$ 101010⟧ results in the literal integer expression 42. If an invalid character is used, a compile-time error is shown. The quasiquoter is defined as follows:

```
bin :: QuasiQuoter
bin = QuasiQuoter { quoteExp = parseBin }
  where
    parseBin :: String → Q Exp
```

```
parseBin s = LitE . IntegerL <$> foldM bindigit 0 s

bindigit :: Integer → Char → Q Integer
bindigit acc '0' = pure (2 * acc)
bindigit acc '1' = pure (2 * acc + 1)
bindigit acc c = fail ("invalid char: " ++ show c)
```

3.5.2 Quasiquotation for pattern matching

Custom quasiquoters allow the DSL user to enter fragments verbatim, bypassing
the syntax of the host language. Pattern matching in general is not suitable for
a custom quasiquoter because it does not really fit in one of the four syntactic
categories for which custom quasiquoter support is available. However, a concrete
use of pattern matching, interesting enough to be beneficial, but simple enough for a
demonstration is the *simple case expression*, a case expression that does not contain
nested patterns and is always exhaustive. They correspond to multi-way conditional
expressions and can thus be converted to DSL constructs straightforwardly (Peyton
Jones, 1987, section 4.4).

In contrast to the binary literal quasiquoter example, we do not hand craft the
parser. The parser combinator library *parsec* is used instead to ease the creation of
the parser (Leijen and Meijer, 2001). First the location of the quasiquoted code is
retrieved using the `location` function that operates in the **Q** monad. This location
is inserted in the parsec parser so that errors are localised in the source code. Then,
the `expr` parser is called that returns an `Exp` in the **Q** monad. The `expr` parser
uses parsec's commodity expression parser primitive `buildExpressionParser`. The
resulting parser translates the string directly into TH's AST data types in the **Q**
monad. The most interesting parser is the parser for the case expression that is
an alternative in the basic expression parser `basic`. A case expression is parsed
when a keyword **case** is followed by an expression that is in turn followed by a
non-empty list of matches. A match is parsed when a pattern (`pat`) is followed
by an arrow and an expression. The results of this parser are fed into the `mkCase`
function that transforms the case into an expression using DSL primitives such as
conditionals, deconstructors and constructor predicates. The above translates to
the following skeleton implementation:

```
expr :: Parser (Q Exp)
expr = buildExpressionParser [···] basic
  where
    basic :: Parser (Q Exp)
    basic =   ···
        <|> mkCase <$ reserved "case" <*> expr
                   <* reserved "of"  <*> many1 match
        <|> ···

    match :: Parser (Q Pat, Q Exp)
    match = (,) <$> pat <* reserved "→ " <*> expr

    pat :: Parser (Q Pat)
```

```
    pat = conP <$> con <*> many var
```

Case expressions are transformed into constructors, deconstructors and constructor predicates, e.g. **case e1 of Cons hd tl → e2; Nil → e3;** is converted to:

```
if' (isList e1)
    (unCons e1 (\hd tl→e2))
    (if' (isNil e1)
        (unNil e1 e3)
        (bottom "Exhausted case"))
```

The `mkCase` (line 1) function transforms a case expression into constructors, deconstructors and constructor predicates. Line 3 first evaluates the patterns. Then the patterns and their expressions are folded using the `mkCase`` function (line 5). While a case exhaustion error is used as the initial value, this is never called since all case expressions are exhaustive. For every case, code is generated that checks whether the constructor used in the pattern matches the constructor of the value using constructor predicates (line 11). If the constructor matches, the deconstructor (line 12) is used to bind all names to the correct identifiers and evaluate the expression. If the constructor does not match, the continuation (`$rest`) is used (line 9).

```
1   mkCase :: Q Exp → [(Q Pat, Q Exp)] → Q Exp
2   mkCase name cases = do
3       pats ← mapM fst cases
4       foldr (uncurry mkCase') ⟦bottom "Exhausted case"⟧
5           (zip pats (map snd cases))
6     where
7       mkCase' :: Pat → Q Exp → Q Exp → Q Exp
8       mkCase' (ConP cons fs) e rest
9           = ⟦if' $pred $then_ $rest⟧
10        where
11          pred  = varE (predicateName cons) `appE` name
12          then_ = ⟦$(varE (deconstructorName cons))
13                  $name $(lamE [pure f | f←fs] e)⟧
```

Finally, with this quasiquotation mechanism we can define our list summation using a case expression. As a byproduct, syntactic cruft such as the special symbols for the operators and calls to `lit` can be removed as well resulting in the following summation implementation:

```
program :: (ListDSL v, DSL v, · · · ) ⇒ v Int
program
    = fun \sum→(\l→ ⟦dsl case l of
        Cons hd tl → hd + sum tl
        Nil        → 0⟧)
    :- sum (cons (lit 38) (cons (lit 4) nil))
```

3.6   Related work

Generic or polytypic programming is a promising technique at first glance for automating the generation of function implementations (Lämmel and Peyton Jones, 2003). However, while it is possible to define a function that works on all first-order types, adding a new function with a new name to the language is not possible. This does not mean that generic programming is not useable for embedding pattern matches. In generic programming, types are represented as sums of products and using this representation it is possible to define pattern matching functions.

For example, Rhiger (2009) showed a method for expressing statically typed pattern matching using typed higher-order functions. If not the host language but the DSL contains higher order functions, the same technique could be applied to port pattern matching to DSLs though using an explicit sums of products representation. Atkey et al. describe embedding pattern matching in a DSL by giving patterns an explicit representation in the DSL by using pairs, sums and injections (Atkey et al., 2009, section 3.3).

McDonell et al. (2022) extends on this idea, resulting in a very similar but different solution to ours. They used the technique that Atkey et al. showed and applied it to deep embedding using the concrete syntax of the host language. The scaffolding—e.g. generating the pairs, sums and injections—for embedding is automated using generics but the required pattern synonyms are generated using TH. The key difference to our approach is that we specialise the implementation for each of the interpretations instead of providing a general implementation of data type handling operations. Furthermore, our implementation does not require a generic function to trace all constructors, resulting in problems with (mutual) recursion.

Young et al. (2021) added pattern matching to a deeply embedded DSL using a compiler plugin. This plugin implements an `externalise :: a → E a` function that allows lifting all machinery required for pattern matching automatically from the host language to the DSL. Under the hood, this function translates the pattern match to constructors, deconstructors, and constructor predicates. The main difference with this work is that it requires a compiler plugin while our metaprogramming approach works on any compiler supporting a metaprogramming system similar to TH.

3.6.1   Related work on Template Haskell

Metaprogramming in general is a very broad research topic and has been around for years already. We therefore do not claim an exhaustive overview of related work on all aspects of metaprogramming. However, we have tried to present most research on metaprogramming in TH. Czarnecki et al. (2004) provide a more detailed comparison of different metaprogramming techniques. They compare staged interpreters, metaprogramming and templating by comparing MetaOCaml, TH and C++ templates. TH has been used to implement related work. They all differ slightly in functionality from our domain and can be divided into several categories.

Generating extra code

Using TH or other metaprogramming systems it is possible to add extra code to your program. The original TH paper showed that it is possible to create variadic functions such as `printf` using TH that would be almost impossible to define without (Sheard and Peyton Jones, 2002). Hammond et al. (2003) used TH to generate parallel programming skeletons. In practice, this means that the programmer selects a skeleton and, at compile time, the code is massaged to suit the pattern and information about the environment is inlined for optimisation.

Polak and Jarosz (2006) implemented automatic GUI generation using TH. Duregård and Jansson (2011) wrote a parser generator using TH and the custom quasiquoting facilities. From a specification of the grammar, given in verbatim using a custom quasiquoter, a parser is generated at compile time. Shioda et al. (2014) used metaprogramming in the D programming language to create a DSL toolkit. They also programmatically generate parsers and an interpretation for either compiling or interpreting the intermediate representation (IR). Blanchette et al. (2022) use TH to simplify the development of Liquid Haskell proofs. Folmer et al. (2022) used TH to synthesize CλaSH (Baaij, 2015) ASTs to be processed. In similar fashion, Materzok (2022) used TH to translate YieldFSM programs to CλaSH.

Optimisation

Besides generating code, it is also possible to analyse existing code and perform optimisations. Yet, this is dangerous territory because unwantedly, the semantics of the optimised program may be slightly different from the original program. For example, Lynagh (2003) implemented various optimisations in TH such as automatic loop unrolling. The compile-time executed functions analyse the recursive function and unroll the recursion to a fixed depth to trade execution speed for program space. Also, O'Donnell (2004) embedded Hydra, a hardware description language, in Haskell utilising TH. Using intensional analysis of the AST, it detects cycles by labelling nodes automatically so that it can generate *netlists*. The authors mention that alternatively this could have be done using a monad but this hampers equational reasoning greatly, which is a key property of Hydra. Finally, Viera et al. (2018) present a way of embedding attribute grammars in Haskell in a staged fashion. Checking several aspects of the grammar is done at compile time using TH while other safety checks are performed at runtime.

Compiler extension

Sometimes, expressing certain functionalities in the host languages requires a lot of boilerplate, syntax wrestling, or other pains. Metaprogramming can relieve some of this stress by performing this translation to core constructs automatically. For example, implementing generic—or polytypic— functions in the compiler is a major effort. Norell and Jansson (2004) used TH to implement the machinery required to implement generic functions at compile time. Adams and DuBuisson (2012) also explores implementing generic programming using TH to speed things

up considerably compared to regular generic programming. Clifton-Everest et al.
(2014) use TH with a custom quasiquoter to offer skeletons for workflows and
embed foreign function interfaces in a DSL. Eisenberg and Stolarek (2014) showed
that it is possible to programmatically lift some functions from the function domain
to the type domain at compile time, i.e. type families. Furthermore, Seefried
et al. (2004) argued that it is difficult to do some optimisations in eDSLs and
that metaprogramming can be of use there. They use TH to change all types to
unboxed types, unroll loops to a certain depth and replace some expressions by
equivalent more efficient ones. Torrano and Segura (2005) showed that it is possible
to use TH to perform a strictness analysis and perform let-to-case translation.
Both applications are examples of compiler extensions that can be implemented
using TH. Another example of such a compiler extension is shown by Gill (2009).
They created a meta level DSL to describe rewrite rules on Haskell syntax that are
applied on the source code at compile time.

Quasiquotation

By means of quasiquotation, the host language syntax that usually seeps through
the embedding can be hidden. The original TH quasiquotation paper (Mainland,
2007) shows how this can be done for regular expressions, not only resulting in
a nicer syntax but syntax errors are also lifted to compile time instead of run
time. Also, Kariotis et al. (2008) used TH to automatically construct monad stacks
without having to resort to the monad transformers library which requires advanced
type system extensions.

Najd et al. (2016) uses the compile time to be able to do normalisation for a DSL,
dubbing it quoted DSLs (QDSLs). They utilise the quasiquotation facilities of TH to
convert Haskell DSL code to constructs in the DSL, applying optimisations such as
eliminating lambda abstractions and function applications along the way. Egi et al.
(2022) extended Haskell to support non-free data type pattern matching—i.e. data
type with no standard form, e.g. sets, graphs—using TH. Using quasiquotation,
they make a complicated embedding of non-linear pattern matching available
through a simple lens.

Typed Template Haskell

Typed Template Haskell (TTH) is a very recent extension/alternative to normal
TH (Pickering et al., 2019; Xie et al., 2022). Whereas in TH you can manipulate
arbitrary parts of the syntax tree, add top-level splices of data types, definitions
and functions, in TTH the programmer can only splice expressions but the AST
fragments representing the expressions are well-typed by construction instead of
untyped.

Pickering et al. (2020) implemented staged compilation for the *generics-sop*
(de Vries and Löh, 2014) generics library to improve the efficiency of the code using
TTH. Willis et al. (2020) used TTH to remove the overhead of parsing combinators.

3.7   Discussion

This chapter aims to be twofold, first, it shows how to inherit data types in a DSL as first-class citizens by generating the boilerplate at compile time using TH. Secondly, it introduces the reader to TH by giving an overview of the literature in which TH is used and provides a gentle introduction by explaining the case study.

FP languages are especially suitable for embedding DSLs but adding user-defined data types is still an issue. The tagless-final style of embedding offers great modularity, extensibility and flexibility. However, user-defined data types are awkward to handle because the built-in operations on them—construction, deconstruction and constructor tests—are not inherited from the host language. We showed how to create a TH function that will splice the required class definitions and view instances. The code dataset also contains an implementation for defining field selectors and provides an implementation for a compiler (see episode III). Furthermore, by writing a custom quasiquoter, pattern matches in natural syntax can be automatically converted to the internal representation of the DSL, thus removing the syntax burden of the facilities. The use of a custom quasiquoter does require the DSL programmer to write a parser for their DSL, i.e. the parser is not inherited from the host language as is often the case in an embedded DSL. However, by making use of modern parser combinator libraries, this overhead is limited and errors are already caught at compilation.

3.7.1   Future work

For future work, it would be interesting to see how generating boilerplate for user-defined data types translates from shallow embedding to deep embedding. In deep embedding, the language constructs are expressed as data types in the host language. Adding new constructs, e.g. constructors, deconstructors, and constructor tests, for the user-defined data type therefore requires extending the data type. Techniques such as data types à la carte (Swierstra, 2008) and open data types (Löh and Hinze, 2006) show that it is possible to extend data types orthogonally but whether metaprogramming can still readily be used is something that needs to be researched. It may also be possible to implemented (parts) of the boilerplate generation using TTH (see section 3.6.1) to achieve more confidence in the type correctness of the implementation.

Another direction of research is to try to find the limits of this technique regarding richer data type definitions. It would be interesting to see whether it is possible to apply the technique on data types with existentially quantified type variables or full-fledged generalised ADTs (Hinze, 2003). It is not possible to straightforwardly lift the deconstructors to type classes because existentially quantified type variables will escape. Rank-2 polymorphism offers tools to define the types in such a way that this is not the case anymore. However, implementing compiling views on the DSL is complicated because it would require inventing values of an existentially quantified type variable to satisfy the type system which is difficult. Finally, having to write a parser for the DSL is extra work. Future research could determine whether it is possible to generate this using TH as well.

Episode II:

Orchestrating the Internet of Things using Task-Oriented Programming

Chapter 4

An introduction to edge device programming

This chapter introduces the monograph. It compares traditional edge device programming to TOP by:

- introducing edge device programming;
- showing how to create the *Hello World!* application for microcontrollers using Arduino and mTask;
- extending the idea to cooperative multitasking, uncovering problems using Arduino that do not exist in mTask;
- and concluding with a reading guide for the remainder of the monograph.

The edge layer of IoT systems predominantly consists of microcontrollers. Microcontrollers are tiny computers designed specifically for embedded applications. They differ significantly from regular computers in many aspects. For example, they are much smaller; only have a fraction of the memory and processor speed; and run on different architectures. Furthermore, they have much more energy-efficient sleep modes, and support connecting and interfacing with peripherals such as sensors and actuators. To illustrate the difference in characteristics, table 4.1 compares the hardware properties of a typical laptop to the characteristics two popular microcontrollers. As a consequence of these differences, development for microcontrollers is unlike development for traditional computers. Programming microcontrollers requires an elaborate multistep toolchain of compilation, linkage, binary image creation, and burning this image onto the flash memory of the microcontroller in order to run a program. Furthermore, as there is no OS to coordinate multiple tasks running at the same time, the software is usually written as a cyclic executive. Hence, all tasks must be manually combined into a single program.

Table 4.1: Hardware characteristics of a laptop and two typical microcontrollers.

	Laptop	Atmega328P	ESP8266
CPU speed	2 GHz to 4 GHz	16 MHz	80 MHz or 160 MHz
№ cores	4 to 8	1	1
Storage	1 TiB	32 KiB	0.5 MiB to 4 MiB
RAM	4 GiB to 16 GiB	2 KiB	160 KiB
Power	50 W to 100 W	0.13 mW to 250 mW	0.1 mW to 350 mW
Size	$\pm 1060\,\mathrm{cm}^3$	$\pm 7.5\,\mathrm{cm}^3$	$\pm 1.1\,\mathrm{cm}^3$
Display	$1920 \times 1080 \times 24$	$1 \times 1 \times 1$	$1 \times 1 \times 1$
Price	€1500	€3	€4

All microcontroller models require their own vendor-provided drivers, hardware abstraction layer, compilers and RTSs. To structure this jungle of tools, platforms exist that provide an abstraction layer over the low-level toolchains. An example of this is the Arduino environment.[1] Originally it was designed for the in-house developed open-source hardware with the same name, but the setup allows porting to many architectures by vendor-provided *cores*. This set of tools is specifically designed for education and prototyping and hence used here to illustrate traditional microcontroller programming. It consists of an IDE containing toolchain automation, a dialect of C/C++, and libraries providing an abstraction layer for microcontroller behaviour. With Arduino, the programmer can program multiple types of microcontrollers using a single language. Using the IDE and toolchain automation, code can be executed easily on many types of microcontrollers with a single press of a button.

4.1   TOP for the IoT

TOP is a programming paradigm that allows multi-tier interactive systems to be generated from a single declarative source (see section 1.4). An example of a TOP system is iTask, a general-purpose TOP language for programming interactive distributed web applications. Such web applications often form the core of the topmost two layers of IoT applications: the presentation and application layer. Furthermore, IoT edge devices are typically programmed with similar workflow-like programs for which TOP is very suitable. Directly incorporating the perception layer, and thus edge devices, in iTask however is not straightforward. All iTask applications carry the weight of multi-user TOP programs that can generically generate webpages, communication, and storage for all data types in the program. As a result, the iTask system targets relatively fast and hence energy-hungry systems with large amounts of RAM and a speedy connection. Edge devices in IoT systems are typically slow but energy efficient and do not have the memory to run the naturally heap-heavy feature-packed functional programs that iTask

---

[1] https://www.arduino.cc, accessed on: 19th December, 2022

programs are. The mTask system bridges this gap by providing a domain-specific TOP language for IoT edge devices. Domain-specific knowledge is embedded in the language and execution platform and unnecessary features for edge devices are removed to drastically lower the hardware requirements. Programs in mTask are written in the mTask DSL, a TOP language that offers a similar abstraction level as iTask. Tasks in mTask operate as if they are iTask tasks, their task value is observable by other tasks, and they can share data using iTask SDSs. This allows for programming entire IoT systems from a single abstraction level, source code, and programming paradigm.

4.2 Hello world!

Traditionally, the first program that one writes when trying a new language is the so-called *Hello World!* program. This program has the single task of printing the text *Hello World!* to the screen and exiting again. It helps the programmer to become familiarised with the syntax of the language and to verify that the toolchain and runtime environment are working. Microcontrollers usually do not come with screens in the traditional sense. Nevertheless, almost always there is a built-in 1 pixel screen with a *1* bit color depth, namely the on-board LED. The *Hello World!* equivalent for microcontrollers blinks this LED.

Creating a blink program using C/C++ and the Arduino libraries result in the code seen in listing 4.1. Arduino programs are implemented as cyclic executives and hence, each program defines a `setup` and a `loop` function. The `setup` function is executed only once on boot, the `loop` function is continuously called afterwards and contains the event loop. In between the executions of the `loop` function, system and maintenance code is executed. In the blink example, the `setup` function only contains code for setting the GPIO pin to the correct mode. The `loop` function alternates the state of the pin representing the LED between `HIGH` and `LOW`, turning the LED off and on respectively. In between, it waits *500* ms so that the blinking is actually visible for the human eye.

```cpp
void setup() {
    pinMode(D2, OUTPUT);
}
void loop() {
    digitalWrite(D2, HIGH);
    delay(500);
    digitalWrite(D2, LOW);
    delay(500);
}
```

Listing (C++) 4.1: Blinking an LED.

4.2.1 Blinking the LED in mTask

Naively translating the traditional blink program to mTask can be done by simply substituting syntax as seen in listing 4.2. E.g. `digitalWrite` becomes `writeD`,

literals are prefixed with `lit`, and `pinMode` becomes `declarePin`. In contrast to the imperative C++ dialect, mTask is a TOP language and therefore there is no such thing as a loop, only task combinators to combine tasks. The task is not the single cyclic executive and therefore consists of just a main expression. The task resulting from the main expression is continuously executed by the RTS. To simulate a loop, the `rpeat` task combinator is used as this task combinator executes the argument task and, when stable, reinstates it. The body of the `rpeat` task contains a task that writes to the pins and waits in between. The tasks are connected using the sequential `>>|.` combinator that for all current intents and purposes executes the tasks after each other.

```
blinkTask :: Main (MTask v ()) | mtask v
blinkTask = declarePin D2 PMOutput \ledPin→
    {main = rpeat (
            writeD ledPin true
        >>|. delay (lit 500)
        >>|. writeD ledPin false
        >>|. delay (lit 500))
    }
```
Listing (Clean) 4.2: Blinking the LED using the `rpeat` combinator.

The mTask DSL is hosted in a full-fledged FP language. It is therefore also possible to define the blinking behaviour as a function. Listing 4.3 shows this more natural translation. The `main` expression is a call to the `blink` mTask function parametrised with the state. The `blink` function first writes the current state to the LED, waits for the specific time, and calls itself recursively with the inverse of the state, resulting in the blinking behaviour. Creating recursive functions like this is not possible in the Arduino language because the program would run out of stack quickly and combining multiple tasks defined like this would be very difficult.

```
blinkTask :: Main (MTask v ()) | mtask v
blinkTask = declarePin D2 PMOutput \ledPin→
    fun \blink = (\st→
            writeD ledPin st
        >>|. delay (lit 500)
        >>|. blink (Not st))
    In {main = blink true}
```
Listing (Clean) 4.3: Blinking the LED using a function.

4.3   Multitasking

Now say that we want to blink multiple blinking patterns on different LEDs concurrently. For example, blink three LEDs connected to GPIO pins *1*, *2* and *3* at intervals of *500* ms, *300* ms and *800* ms. Intuitively, you would want to lift the blinking behaviour to a function in order to minimise duplicate code, and increase modularity by calling this function three times with different parameters as shown in listing 4.4.

```cpp
void setup () { ... }
void blink(int pin, int wait) {
    digitalWrite(pin, HIGH);
    delay(wait);
    digitalWrite(pin, LOW);
    delay(wait);
}
void loop() {
    blink (D1, 500);
    blink (D2, 300);
    blink (D3, 800);
}
```

Listing (C++) 4.4: Naive approach to multiple blinking patterns.

```cpp
long led1 = 0,    led2 = 0,    led3 = 0;
bool st1 = false, st2 = false, st3 = false;

void setup () { ... }
void blink(int pin, int interval, long *lastrun, bool *st) {
    if (millis() - *lastrun > interval) {
        digitalWrite(pin, *st = !*st);
        *lastrun += interval;
    }
}
void loop() {
    blink(D1, 500, &led1, &st1);
    blink(D2, 300, &led2, &st1);
    blink(D3, 800, &led3, &st1);
}
```

Listing (C++) 4.5: Threading three blinking patterns.

Unfortunately, this does not work because the `delay` function blocks all other execution. The resulting program blinks the LEDs after each other instead of at the same time. To overcome this, it is necessary to slice up the blinking behaviour in small fragments and interleave them manually (Feijs, 2013).

Listing 4.5 shows how three different blinking patterns could be implemented in Arduino using the slicing method. If we want the blink function to be a separate parametrisable function we need to explicitly provide all references to the required global state. Furthermore, the `delay` function can not be used and polling `millis` is required. The `millis` function returns the number of milliseconds that have passed since the boot of the microcontroller. If the delay passed to the `delay` function is long enough, the firmware may decide to put the processor in sleep mode, reducing the power consumption drastically. When polling `millis` is used, this therefore potentially affects power consumption since the processor is busy

looping all the time, not knowing when to go to sleep. Manually combining tasks into a single modular program is very error-prone, requires a lot of pointer juggling, and generally results into spaghetti code. Furthermore, it is very difficult to represent dependencies between threads. Often state machines have to be explicitly programmed and merged by hand to achieve this. In the simple case of blinking three LEDs according to fixed intervals, it is possible to calculate the delays in advance using static analysis and generate the appropriate `delay` calls. Unfortunately, this is very hard when for example the blinking patterns are determined at runtime.

4.3.1   Multitasking in mTask

In mTask, expressions are eagerly evaluated in an interpreter and tasks are executed by small-step rewrite rules. In between these rewrite steps, other tasks are executed and communication is handled. Consequently, and in contrast to Arduino, the `delay` task in mTask does not block the execution. It has no observable value until the target waiting time has passed, and is thence *stable*. As there is no global state, the function is parametrised with the current status, the pin to blink and the waiting time. With a parallel combinator, tasks are executed seemingly at the same time, i.e. their very short small-step reduction steps are interleaved. Therefore, blinking three different blinking patterns is as simple as combining the three calls to the `blink` function with their arguments as seen in listing 4.6.

```
blinktask :: MTask v () | mtask v
blinktask =
    declarePin D1 PMOutput \d1→
    declarePin D2 PMOutput \d2→
    declarePin D3 PMOutput \d3→
    fun \blink = (\(st, pin, wait)→
            delay wait
        >>|. writeD pin st
        >>|. blink (Not st, pin, wait))
    In {main =   blink (true, d1, lit 500)
            .||. blink (true, d2, lit 300)
            .||. blink (true, d3, lit 800)
    }
```

Listing (Clean) 4.6: Threading three blinking patterns.

4.4   Conclusion and reading guide

This chapter introduced traditional edge device programming and programming edge devices using mTask. The edge layer of IoT systems is powered by micro-controllers. Microcontrollers have significantly different characteristics to regular computers. Programming them happens through compiled firmwares using low-level imperative programming languages. Due to the lack of an OS, writing applications that perform multiple tasks at the same time is error-prone, becomes complex,

and requires a lot of boilerplate such as manual scheduling code. With the mTask system, a TOP programming language for IoT edge devices, this limitation can be overcome. Since a lot of domain-specific knowledge is built into the language and RTS, the hardware requirements can be kept relatively low while maintaining a high abstraction level. Tasks in mTask are high-level specifications of the work that needs to be done, they can be combined using task combinators, and share data using SDSs with each other and with the server. Furthermore, the programs are automatically integrated with iTask, a TOP system for creating interactive distributed web applications, allowing for data sharing, task coordination, and dynamic construction of tasks over all layers of an IoT system.

The following chapters of this monograph thoroughly introduce all aspects of the mTask system. First, the language setup and interface are shown in chapter 5. Chapter 6 shows the integration of mTask and iTask. Then, chapter 7 provides the implementation of the DSL, the compilation schemes, instruction set, and details on the interpreter. Chapter 8 explains all green computing aspects of mTask, i.e. task scheduling and processor interrupts. Finally, chapter 9 concludes, discusses related work, and provides a short history of mTask.

Chapter 5

The mTask language

This chapter introduces the TOP language mTask by:

- introducing class-based shallow embedding and the setup of the mTask language;
- describing briefly the various interpretations;
- demonstrating how the type system is leveraged to enforce all constraints;
- showing the language interface for expressions, datatypes, and functions;
- and explaining the tasks, task combinators, and SDSs.

Regular FP and TOP languages do not run on resource-constrained edge devices. A DSL is therefore used as the basis of the mTask system, a complete TOP programming environment for programming microcontrollers. It is implemented as an eDSL in Clean using class-based—or tagless-final—embedding. This means that the language interface, i.e. the language constructs, are a collection of type classes. Interpretations of this interface are data types implementing these classes. Due to the nature of this embedding technique, it is possible to have multiple interpretations for programs written in the mTask language. Furthermore, this particular type of embedding has the property that it is extensible both in language constructs and in interpretations. Adding a language construct is as simple as adding a type class. Adding an interpretation is done by creating a new data type and providing implementations for the various type classes.

In order to reduce the hardware requirements for devices running mTask programs, several measures have been taken that are reflected in the language design. Programs in mTask are written in the mTask DSL, separating them from the host iTask program. This allows the tasks to be constructed at compile time in order to tailor-make them for the specific work requirements. Furthermore, the mTask language is restricted: there are no recursive data structures, no higher-order

functions, strict evaluation, and functions and objects can only be declared at the top level.

5.1 Class-based shallow embedding

The mTask language is implemented as a class-based shallow, also known as tagless-final, embedded DSL (see also section 3.2). In order to demonstrate the technique, it is first illustrated by showing the very simple language of literal values. This language interface can be described using a single type constructor class with a single function `lit`. This function is for lifting values, when it has a `toString` instance, from the host language to our new DSL. The type variable `v` of the type class represents the view on the language, the interpretation.

```
class literals v where
    lit :: a → v a | toString a
```

Providing an evaluator is straightforward as can be seen in the following listing. The evaluator is just a box holding a value of the computation, but it can also be something more complex such as a monadic computation.

```
:: Eval a = Eval a

runEval :: (Eval a) → a
runEval (Eval a) = a

instance literals Eval where
    lit a = Eval a
```

Extending the language with a printer is done by defining a new data type and providing instances for the type constructor classes. The printer shown below only stores a printed representation and hence the type variable is just a phantom type:

```
:: Printer a = Printer String

runPrinter :: (Printer a) → String
runPrinter (Printer a) = a

instance literals Printer where
    lit a = Printer (toString a)
```

Adding language constructs happens by defining new type classes and giving implementations for the interpretations. The following listing adds an addition construct to the language and shows the implementations for the evaluator and printer.

```
class addition v where
    add :: v a → v a → v a | + a

instance addition Eval where
    add (Eval l) (Eval r) = Eval (l + r)
```

```
instance addition Printer where
    add (Printer l) (Printer r) = Printer ("(" +++ l +++ "+" +++ r +++ ")")
```

Terms in our toy language can be overloaded in their interpretation, they are just an interface. For example, $1 + 5$ is written as `add (lit 1) (lit 5)` and has the type `v Int | literals, addition v`. However, due to the way let-polymorphism is implemented in most functional languages, it is not always straightforward to use multiple interpretations in one function. Creating such a function, e.g. one that both prints and evaluates an expression, requires rank-2 polymorphism (see listing 5.4).

5.2   Types

The mTask language is a tagless-final eDSL as well. As it is shallow embedded, the types of the terms in the language can be constrained by type classes. Types in the mTask language are expressed as types in the host language, to make the language type safe. However, not all types in the host language are suitable for microcontrollers that may only have *2* KiB of RAM, so class constraints are added to the DSL functions. Table 5.1 shows the mapping from Clean types to C/C++ types. The most used class constraint is the `type` class collection containing functions for serialisation, printing, iTask constraints, *&c* (Plasmeijer et al., 2021, section 6.9). Most of these functions are automatically derivable using generic programming but can be specialised when needed. An even stronger restriction is defined for types that have a stack representation. This `basicType` class has instances for many Clean basic types such as `Int`, `Real` and `Bool`. These class constraints for values in mTask are omnipresent in all functions and therefore usually omitted for brevity and clarity.

Table 5.1: Translation from Clean/mTask data types to C/C++ datatypes.

Clean/mTask	C/C++	№bits
Bool	**bool**	16
Char	**char**	16
Int	int16_t[1]	16
Real	**float**	32
:: Long	int32_t	32
:: T = A | B | C	**enum**	16

Listing 5.1 contains the definitions for the auxiliary types and type constraints (such as `type` and `basicType`) that are used to construct mTask expressions.

```
class type t | iTask, ··· , fromByteCode, toByteCode t
class basicType t | type t where ···
```

Listing (Clean) 5.1: Classes and class collections for the mTask language.

---

[1]In Arduino C/C++ this usually equals a **long**.

The mTask language interface consists of a core collection of type classes bundled in the type class **class mtask** (see listing 5.2). Every interpretation of mTask terms implements the type classes in the `mtask` class collection.

```clean
class mtask v | expr, ··· , int, real, long v
```
Listing (Clean) 5.2: Class collection for the mTask language.

Peripheral, SDS, and function definitions are always defined at the top level of mTask programs. This is enforced by the `Main` type. Most top level definitions are defined using HOAS. To make their syntax friendlier, the **In** type—an infix tuple—is used to combine these top level definitions. To illustrate the structure of mTask programs, listing 5.3 shows a skeleton of a program.

```clean
// From the mTask library
:: Main a = { main :: a }
:: In a b = (In) infix 0 a b

someTask :: MTask v Int | mtask v & lowerSds v & sensor1 v & ···
someTask =
    sensor1 config1 \sns1→
    sensor2 config2 \sns2→
       sds      \s1   = initialValue
    In lowerSds \s2   = someiTaskSDS
    In fun      \fun1 = ( \(a0, a1)→··· )
    In fun      \fun2 = ( \a→··· )
    In { main = mainexpr }
```
Listing (Clean) 5.3: Auxiliary types and example task in the mTask language.

Expressions in the mTask language are usually overloaded in their interpretation (`v`). In Clean, all free variables in a type are implicitly universally quantified. In order to use the mTask expressions with multiple interpretations, rank-2 polymorphism is required (Odersky and Läufer, 1996). Listing 5.4 shows an example of a function that simulates an mTask expression while showing the pretty printed representation in parallel. Providing a type for the `simulateAndPrint` function is mandatory as the compiler cannot infer the type of rank-2 polymorphic functions (Plasmeijer et al., 2021, section 3.7.4).

```clean
simulateAndPrint :: (A.v: Main (MTask v a) | mtask v) → Task a | type a
simulateAndPrint mt =
        simulate mt
    -|| Hint "Current task:" @>> viewInformation [] (showMain mt)
```
Listing (Clean) 5.4: Rank-2 polymorphism to allow multiple interpretations.

5.3   Expressions

This section shows all mTask language constructs for expressions. Listing 5.5 shows the `expr` class containing the functionality to: lift values from the host language to the mTask language (`lit`); perform numeric and boolean arithmetics;

do comparisons; and perform conditional execution. For every common boolean and arithmetic operator in the host language, an mTask variant is present. The operators are suffixed by a period to not clash with the built-in operators in Clean.

```clean
class expr v where
    lit :: t → v t | type t
    (+.) infixl 6 :: (v t) (v t) → v t | basicType, +, zero t
    ...
    (&.) infixr 3 :: (v Bool) (v Bool) → v Bool
    (|.) infixr 2 :: (v Bool) (v Bool) → v Bool
    Not          :: (v Bool) → v Bool
    (==.) infix 4 :: (v a) (v a) → v Bool | Eq, basicType a
    ...
    If :: (v Bool) (v t) (v t) → v t | type t
```
Listing (Clean) 5.5: The mTask class for expressions.

Conversion to and fro data types is available through the overloaded functions `int`, `long` and `real`. These functions convert the argument to the respective type similar to casting in C. For most interpretations, there are instances of these classes for all numeric types.

```clean
class int  v a :: (v a) → v Int
class real v a :: (v a) → v Real
class long v a :: (v a) → v Long
```
Listing (Clean) 5.6: Type conversion functions in mTask.

Values from the host language must be explicitly lifted to the mTask language using the `lit` function. For convenience, there are many lower-cased macro definitions for often-used constants such as `true :== lit True`, `false :== lit False`.

Listing 5.7 shows some examples of expressions in the mTask language. Since they are only expressions, there is no need for a `Main`. `e0` defines the literal 42, `e1` calculates the literal 42.0 using real numbers and uses a type conversion. `e2` compares `e0` and `e1` as integers and if they are equal it returns $e2/2$ and `e0` otherwise.

```clean
e0 :: v Int | expr v
e0 = lit 42

e1 :: v Real | expr v
e1 = lit 38.0 +. real (lit 4)

e2 :: v Int | expr v
e2 = If (e0 ==. int e1)
    (int e1 /. lit 2) e0
```
Listing (Clean) 5.7: Example mTask expressions.

The mTask language is shallow embedded in Clean and the terms are constructed and hence compiled at run time. This means that mTask programs can also be tailor-made at run time using Clean functions, maximising the linguistic reuse (Krishnamurthi, 2001). The `approxEqual` function in listing 5.8 is an example of this. It performs a simple approximate equality—admittedly not taking into account all

floating point peculiarities. When calling `approxEqual` in an mTask expression, the resulting code is inlined.

```clean
approxEqual :: (v Real) (v Real) (v Real) → v Bool | expr v
approxEqual x y eps = x ==. y
    |.  If (x >. y)
        (x -. y <. eps)
        (y -. x <. eps)
```
<div align="center">Listing (Clean) 5.8: Approximate equality in mTask.</div>

5.3.1 Data types

Most of the fixed-size basic types from Clean are mapped on mTask types (see table 5.1). However, it is useful to have access to compound types as well. All types in mTask have a fixed-size representation on the stack, so sum types are not (yet) supported. It is possible to lift any types, e.g. tuples, using the `lit` function as long as they have instances for the required type classes. However, you cannot do anything with values of the types besides passing them around. To be able to use types as first-class citizens, constructors, and field selectors or deconstructors are required (see chapter 3). Listing 5.9 shows the scaffolding required for supporting tuples in mTask. Besides the constructors and field selectors, there is also a helper function available that transforms a function from a tuple of mTask expressions to an mTask expression of a tuple, a deconstructor. Examples of mTask programs using tuples are seen later in section 5.3.2.

```clean
class tupl v where
    tupl :: (v a) (v b) → v (a, b) | type a & type b
    first :: (v (a, b)) → v a | type a & type b
    second :: (v (a, b)) → v b | type a & type b

    tupopen :: ((v a, v b) → v c) → ((v (a, b)) → v c)
    tupopen f :== \v→f (first v, second v)
```
<div align="center">Listing (Clean) 5.9: Tuple constructor and field selectors in mTask.</div>

5.3.2 Functions

Adding functions to the language is achieved by one type class in the mTask DSL. By using HOAS, both the function definitions and the calls to the functions are controlled by the DSL (Chlipala, 2008; Pfenning and Elliott, 1988). The mTask language enforces all functions to be first-order and forbids partial function application in order to reduce memory use and code size. These restrictions are enforced by using a multi-parameter type class with two parameters instead of a type class with one type variable. The first parameter represents the shape of the arguments, the second parameter the interpretation. An instance is provided for each function arity instead of providing an instance for all possible arities to enforce that all functions are first order. By using argument tuples to represent the arity of the function, it is not possible to create partial function applications. The

```
class fun a v :: ((a → v s) → In (a → v s) (Main (MTask v u)))
    → Main (MTask v u)

instance fun () Show where ···
instance fun (Show a) Show | type a where ···
instance fun (Show a, Show b) Show | type a, type b where ···
instance fun (Show a, Show b, Show c) Show | type a, ··· where ···
···
```

<div align="center">Listing (Clean) 5.10: Functions in mTask.</div>

definition of the type class and some instances for the pretty printer are shown in
listing 5.10.

Deriving how to define and use functions from the type is quite a challenge even
though the resulting syntax is made easier using the infix type **In**. Splitting out
the function definition for each single arity means that for every function arity and
combination of arguments, a separate class constraint is required. Many of the often
used functions signatures are in the `mtask` class constraint collection. Listing 5.11
show some examples of functions to illustrate the syntax. The `factorial` function
shows a recursive version of the factorial function. The `factorialtail` function
is a tail-call optimised version of the above. It also illustrates a manually added
class constraint, as they are required when functions are used that have signatures
not present in the `mtask` class collection. Zero-arity functions are always called
with unit as an argument, which is shown in the `zeroarity` function. Finally, the
`swapTuple` function shows an example of a tuple being swapped using the `tupopen`
macro (see listing 5.9).

```
factorial :: Main (v Int) | mtask v
factorial =
    fun \fac = (\i→If (i <. lit 1)
        (lit 1)
        (i *. fac (i -. lit 1)))
    In {main = fac (lit 5) }

factorialtail :: Main (v Int) | mtask v & fun (v Int, v Int) v
factorialtail =
        fun \facacc = (\(acc, i)→If (i <. lit 1)
            acc
            (fac (acc *. i, i -. lit 1)))
    In fun \fac = (\i→facacc (lit 1, i))
    In {main = fac (lit 5) }

zeroarity :: Main (v Int) | mtask v
zeroarity =
        fun \fourtytwo = (\()→lit 42)
    In fun \add = (\(x, y)→x +. y)
    In {main = add (fourtytwo (), lit 9)}
```

```
swapTuple :: Main (v (Int, Bool)) | mtask v
swapTuple =
        fun \swap = (tupopen \(x, y)→tupl y x)
    In {main = swap (tupl true (lit 42)) }
```

Listing (Clean) 5.11: Examples of various functions in mTask.

5.4   Tasks and task combinators

This section describes the task language of mTask. TOP languages are programming languages enriched with tasks. Tasks represent abstract pieces of work and can be combined using combinators. Creating tasks is done by evaluating expressions. The result of an evaluated task expression is called a task tree, a run time representation of a task. In order to evaluate a task, the resulting task tree is *rewritten* using small-step reduction, i.e. similar to rewrite systems, they perform a bit of work, step by step. With each step, a task value is yielded that is observable by other tasks and can be acted upon.

The implementation in the mTask RTS for task execution is shown in chapter 7. The following sections show the definitions of the functions for creating tasks. They also show the semantics of tasks: their observable value in relation to the work that the task represents. The task language of mTask is divided into three categories:

Basic tasks are the leaves in the task trees. In most TOP systems, the basic tasks are called editors, modelling the interactivity with the user. In mTask, there are no editors in that sense. Editors in mTask model the interaction with the outside world through peripherals such as sensors and actuators.

Task combinators provide a way of describing the workflow or collaboration. They combine one or more tasks into a compound task.

SDSs are typed references to shared memory in mTask. The data is available for tasks using many-to-many communication. Reading, writing and updating SDSs is an atomic operation on the task level.

As mTask is integrated with iTask, a stability distinction is made for task values just as in iTask. A task in mTask is denoted by the DSL type synonym shown in listing 5.12. A task is an expression of the type `TaskValue a` in interpretation v.

```
:: MTask v a :== v (TaskValue a)

// From the iTask library
:: TaskValue a
    = NoValue
    | Value a Bool
```

Listing (Clean) 5.12: Task type in mTask.

5.4.1   Basic tasks

The mTask language contains interactive and non-interactive basic tasks. As mTask is integrated in iTask, the same notion of stability is applied to the task values. Task values have either *no value*, or are *unstable* or *stable* (see figure 1.4). Once a task yields a stable value, it does not change anymore. The most rudimentary non-interactive basic tasks in the task language of mTask are `rtrn` and `unstable`. They lift the value from the mTask expression language to the task domain either as a stable or unstable value. There is also a special type of basic task for delaying execution. The `delay` task—parametrised by a number of milliseconds—yields an unstable value while the time has not passed. Once the specified time has passed, the time the task overshot the target time is yielded as a stable task value. See section 5.4.2 for an example task using `delay`.

```clean
class rtrn v :: (v t) → MTask v t
class unstable v :: (v t) → MTask v t
class delay v :: (v n) → MTask v n | long v n
```

Listing (Clean) 5.13: Non-interactive basic tasks in mTask.

Peripherals

In order for the edge device to interact with the environment, peripherals such as sensors and actuators are employed. Some peripherals are available on the microcontroller package, others are connected with wires using protocols such as I^{2}C. For every supported sensor or actuator, basic tasks are available that allow interaction with the specific peripheral. The type classes for these tasks are not included in the `mtask` class collection as not all devices nor all language interpretations support every peripheral connected.

An example of a built-in peripheral is the GPIO system. This array of digital and analogue pins is controlled through software. GPIO access is divided into three classes: analogue I/O, digital I/O and pin mode configuration (see listing 5.14). For all pins and pin modes, an ADT is available that enumerates the pins. The analogue GPIO pins of a microcontroller are connected to an analog-to-digital converter (ADC) that translates the measured voltage to an integer. Digital GPIO pins of a microcontroller report either a high or a low value. Both analogue and digital GPIO pins can be read or written to, but it is advised to set the pin mode accordingly. The `pin` type class allows functions to be overloaded in either the analogue or digital pins, e.g. analogue pins can be considered digital pins as well.

For digital GPIO interaction, the `dio` type class is used. The first argument of the functions in this class is overloaded, i.e. it accepts either an `APin` or a `DPin`. Analogue GPIO tasks are bundled in the `aio` type class. GPIO pins usually operate according to a certain pin mode that states whether the pin acts as an input pin, an input with an internal pull-up resistor or an output pin. This setting can be set using the `pinMode` class by hand or by using the `declarePin` GPIO pin constructor to declare it at the top level. Setting the pin mode is a task that immediately finishes and yields a stable unit value. Writing to a pin is also a task

that immediately finishes, but yields the written value. Reading a pin is a task
that yields an unstable value—i.e. the value read from the actual pin.

```
:: APin = A0 | A1 | A2 | A3 | A4 | A5
:: DPin = D0 | D1 | D2 | D3 | D4 | D5 | D6 | D7 | D8 | D9 | D10 | ···
:: PinMode = PMInput | PMOutput | PMInputPullup
:: Pin = AnalogPin APin | DigitalPin DPin

class pin p :: p → Pin
instance pin APin, DPin

class aio v where
    writeA :: (v APin) (v Int) → MTask v Int
    readA :: (v APin) → MTask v Int

class dio p v | pin p where
    writeD :: (v p) (v Bool) → MTask v Bool
    readD :: (v p) → MTask v Bool | pin p

class pinMode v where
    pinMode :: (v PinMode) (v p) → MTask v () | pin p
    declarePin :: p PinMode ((v p) → Main (v a)) → Main (v a) | pin p
```

Listing (Clean) 5.14: The mTask interface for GPIO access.

Listing 5.15 shows two examples of mTask tasks using GPIO pins. `task1` reads
analogue GPIO pin 3. This is a task that never terminates. `task2` writes the `true`
(Arduino `HIGH`) value to digital GPIO pin 3. This task finishes immediately after
writing to the pin.

```
task1 :: MTask v Int | mtask v
task1 = declarePin A3 PMInput \a3→{main=readA a3}

task2 :: MTask v Int | mtask v
task2 = declarePin D3 PMOutput \d3→{main=writeD d3 true}
```

Listing (Clean) 5.15: GPIO example in mTask.

Peripherals are bundled by their functionality in mTask. For example, list-
ing 5.16 shows the type classes for all supported digital humidity and temperature
(DHT) sensors. Currently, two different types of DHT sensors are supported, the
DHT family of sensors connect through the 1-wire protocol and the *SHT* family
of sensors connected using the I^{2}C protocol. Creating such a `DHT` object is very
similar to creating functions in mTask and uses HOAS to make it type safe. When
provided a configuration and a configuration function, the DHT object can be
brought into scope. For the DHT sensor there are two basic tasks, `temperature` and
`humidity`, that produce a task that yields the observed temperature in °C or the
relative humidity as an unstable value. Other peripherals have similar interfaces,
they are available in appendix B.1.

```
:: DHT //abstract
:: DHTInfo
    = DHT_DHT Pin DHTtype
    | DHT_SHT I2CAddr
:: DHTtype = DHT11 | DHT21 | DHT22
class dht v where
    DHT :: DHTInfo ((v DHT) → Main (v b)) → Main (v b) | type b
    temperature :: (v DHT) → MTask v Real
    humidity :: (v DHT) → MTask v Real

measureTemp :: Main (MTask v Real) | mtask, dht v
measureTemp = DHT (DHT_SHT (i2c 0x36)) \dht→{main=temperature dht}
```

<div align="center">Listing (Clean) 5.16: The mTask interface for DHTs sensors.</div>

5.4.2   Task combinators

Task combinators are used to combine multiple tasks to describe workflows. The mTask language has a set of simpler combinators from which more complicated workflow can be derived. There are three main types of task combinators, namely:

- sequential combinators that execute tasks one after the other, possibly using the result of the left-hand side;
- parallel combinators that execute tasks at the same time, combining the result;
- and miscellaneous combinators that change the semantics of a task—for example a combinator that repeats the child task.

Sequential

Sequential task combination allows the right-hand side expression to *observe* the left-hand side task value. All sequential task combinators are defined in the `step` class and are by default defined in terms of the Swiss Army knife step combinator (`>>*.`, listing 5.17). This combinator has a single task on the left-hand side and a list of *task continuations* on the right-hand side. Every rewrite step, the list of task continuations are tested on the task value. If one of the predicates matches, the task continues with the result of these continuations. Several shorthand combinators are derived from the step combinator. The `>>=.` combinator is a shorthand for the bind operation, if the left-hand side is stable, the right-hand side function is called to produce a new task. The `>>|.` combinator is a shorthand for the sequence operation, if the left-hand side is stable, it continues with the right-hand side task. The `>>~.` and `>>..` combinators are variants of the ones above that ignore the stability and continue on an unstable value as well.

```
class step v | expr v where
    (>>*.) infixl 1 :: (MTask v t) [Step v t u] → MTask v u
    (>>=.) infixl 0 :: (MTask v t) ((v t) → MTask v u) → MTask v u
    (>>|.) infixl 0 :: (MTask v t)          (MTask v u) → MTask v u
```

```
(>>~.) infixl 0 :: (MTask v t) ((v t) → MTask v u) → MTask v u
(>>..) infixl 0 :: (MTask v t)          (MTask v u) → MTask v u

:: Step v t u
   = IfValue    ((v t) → v Bool) ((v t) → MTask v u)
   | IfStable   ((v t) → v Bool) ((v t) → MTask v u)
   | IfUnstable ((v t) → v Bool) ((v t) → MTask v u)
   | Always                                (MTask v u)
```
Listing (Clean) 5.17: Sequential task combinators in mTask.

Listing 5.18 shows an example task containing a step. The `readPinBin` function produces an mTask task that classifies the value of an analogue pin into four bins. It also shows that the nature of embedding allows the host language to be used as a macro language.

```
readPinBin :: Int → Main (MTask v Int) | mtask v
readPinBin lim = declarePin PMInput A2 \a2→
   { main = readA a2 >>*.
      [  IfValue (\x→x <. lim) (\_→rtrn (lit bin))
      \\ lim <- [64,128,192,256]
      &  bin <- [0..]]}
```
Listing (Clean) 5.18: Read an analogue pin and bin the value in mTask.

Parallel

The result of a parallel task combination is a compound task that executes both tasks at the same time. There are two types of parallel task combinators in the mTask language (see listing 5.19).

```
class (.&&.) infixr 4 v :: (MTask v a) (MTask v b) → MTask v (a, b)
class (.||.) infixr 3 v :: (MTask v a) (MTask v a) → MTask v a
```
Listing (Clean) 5.19: Parallel task combinators in mTask.

The conjunction combinator (`.&&.`) combines the result by putting the values from both sides in a tuple. The stability of the task depends on the stability of both children. If both children are stable, the result is stable, otherwise the result is unstable. The disjunction combinator (`.||.`) combines the results by picking the leftmost, most stable task. The semantics of both parallel combinators are most easily described using the Clean functions shown in listings 5.20 and 5.21.

Listing 5.22 gives an example program that uses the parallel task combinator. This task read two pins at the same time, returning when one of the pins becomes high. If the combinator was the `.&&.`, the type would be `MTask v (Bool, Bool)` and the task would only return when both pins are high but not necessarily at the same time.

Repeat

In many workflows, tasks are to be executed repeatedly. The `rpeat` combinator does this by executing the child task until it becomes stable. Once a stable value is

```
con :: (TaskValue a) (TaskValue b)        dis :: (TaskValue a) (TaskValue a)
    → TaskValue (a, b)                        → TaskValue a
con NoValue r = NoValue                    dis NoValue r = r
con l NoValue = NoValue                    dis l NoValue = l
con (Value l ls) (Value r rs)              dis (Value l ls) (Value r rs)
    = Value (l, r) (ls && rs)                  | rs && not ls = Value r rs
                                               | otherwise = Value l ls
```

Listing (Clean) (5.20) Semantics of the conjunction combinator.

Listing (Clean) (5.21) Semantics of the disjunction combinator.

```
task :: MTask v Bool
task =
    declarePin D0 PMInput \d0→
    declarePin D1 PMInput \d1→
    fun \monitor = (\pin→readD pin >>*. [IfValue id rtrn])
    In {main = monitor d0 .||. monitor d1}
```

Listing (Clean) 5.22: Parallel task combinator example in mTask.

observed, the task is reinstated. The functionality of `rpeat` can also be simulated by using functions and sequential task combinators and even made to be stateful as can be seen from the blink example from chapter 4.

```
class rpeat v where
    rpeat :: (MTask v a) → MTask v a
```

Listing (Clean) 5.23: Repeat task combinators in mTask.

Listing 5.24 shows an example of a task that uses the `rpeat` combinator. This resulting task mirrors the value read from analogue GPIO pin `A1` to pin `A2` by constantly reading the pin and writing the result.

```
task :: MTask v Int | mtask v
task = declarePin A1 PMInput  \a1→
       declarePin A2 PMOutput \a2→
       {main = rpeat (readA a1 >>~. writeA a2 >>|. delay (lit 1000))}
```

Listing (Clean) 5.24: Repeat task combinator example in mTask.

5.4.3 Shared data sources

For some collaborations it is cumbersome to only communicate data using task values. SDSs are a safe abstraction over any data that fill this gap. It allows tasks to safely and atomically read, write and update data stores in order to share data with other tasks. SDSs in mTask are by default references to shared memory but can also be references to SDSs in iTask (see section 6.3). There are no combinators

or user-defined SDS types in mTask as there are in iTask. Similar to peripherals and functions, SDSs are defined at the top level with the `sds` function. They are accessed through interaction tasks. The `getSds` task yields the current value of the SDS as an unstable value. This behaviour is similar to the `watch` task in iTask. Writing a new value to an SDS is done using `setSds`. This task yields the written value as a stable result after it is done writing. Getting and immediately setting an SDS is not necessarily an *atomic* operation in mTask because it is possible that another task accesses the SDS in between. To circumvent this issue, `updSds` is available by which an SDS can be atomically updated. The `updSds` task only guarantees atomicity within mTask.

```
:: Sds a // abstract
class sds v where
    sds :: ((v (Sds t)) → In t (Main (MTask v u))) → Main (MTask v u)
    getSds :: (v (Sds t))                    → MTask v t
    setSds :: (v (Sds t)) (v t)              → MTask v t
    updSds :: (v (Sds t)) ((v t) → v t) → MTask v t
```
Listing (Clean) 5.25: SDSs in mTask.

Listing 5.26 shows an example task that uses SDSs. The `count` function takes a pin and returns a task that counts the number of times the pin is observed as high by incrementing the `share` SDS. In the `main` expression, this function is called twice with a different argument. The results are combined using the parallel disjunction combinator (`.||.`). Using a sequence of `getSds` and `setSds` would be unsafe here because the other branch might write their old increment value immediately after writing, effectively missing a count.

```
task :: MTask v Int | mtask v
task =
    declarePin D3 PMInput \d3→
    declarePin D5 PMInput \d5→
       sds \share=0
    In fun \count=(\pin→
           readD pin
        >>* [IfValue id (\_→updSds (\x→x +. lit 1) share)]
        >>| delay (lit 100) // debounce
        >>| count pin)
    In {main=count d3 .||. count d5}
```
Listing (Clean) 5.26: Examples with SDSs in mTask.

5.5   Interpretations

The nature of the mTask DSL embedding allows for multiple interpretations of the terms in the language. The mTask language has interpretations to pretty print, simulate, and generate byte code for terms in the language. There are many other interpretations possible such as static analyses or optimisation. Not all these interpretations are necessarily TOP engines, i.e. not all the interpretations execute the resulting tasks.

5.5.1   Pretty printer

The pretty printer interpretation converts an mTask term to a string representation. As the host language Clean constructs the mTask expressions at run time, it can be useful to show the constructed expression at run time as well. The only function exposed for this interpretation is the `showMain` function (listing 5.27). It runs the pretty printer and returns a list of strings containing the pretty printed result. The pretty printing function does the best it can but obviously cannot reproduce layout, curried functions, and variable names. This shortcoming is illustrated by printing a blink task that contains a function and currying in listing 5.28. The output of this action would be `fun f0 a1 = writeD(D13, a1) >>= \a2.(delay 1000) >>| (f0 (Not a1)) in (f0 True)`

```
:: Show a // from the mTask pretty printing library
showMain :: (Main (Show a)) → [String]
```
Listing (Clean) 5.27: The entry point for the pretty printing interpretation.


```
blinkTask :: Main (MTask v Bool) | mtask v
blinkTask =
    fun \blink=(\state→
        writeD d13 state >>|. delay (lit 500) >>=. blink o Not
    ) In {main = blink true}
```
Listing (Clean) 5.28: Pretty printing interpretation example.

5.5.2   Simulator

In a real microprocessor, it is hard to observe the state and to control the sensors in such a way that the behaviour of interest can be observed. The simulator converts the expression to a ready-for-work iTask simulation to bridge this gap. There is one entry point for this interpretation (see listing 5.29). The task resulting from the `simulate` function presents the user with an interactive simulation environment (see figure 5.2). The simulation allows the user to (partially) execute tasks, control the simulated peripherals, inspect the internal state of the tasks, and interact with SDSs.

```
:: TraceTask a // from the mTask simulator library
simulate :: (Main (TraceTask a)) → [String]
```
Listing (Clean) 5.29: The entry point for the simulation interpretation.

5.5.3   Byte code compiler

The main interpretation of the mTask system is the byte code compiler (`:: BCInterpret a`). This interpretation compiles the mTask term at run time to byte code. With it, and a handful of integration functions, mTask tasks can be executed on microcontrollers and integrated in iTask as if they were regular iTask tasks. Furthermore, with a special language construct, SDSs can be shared between mTask
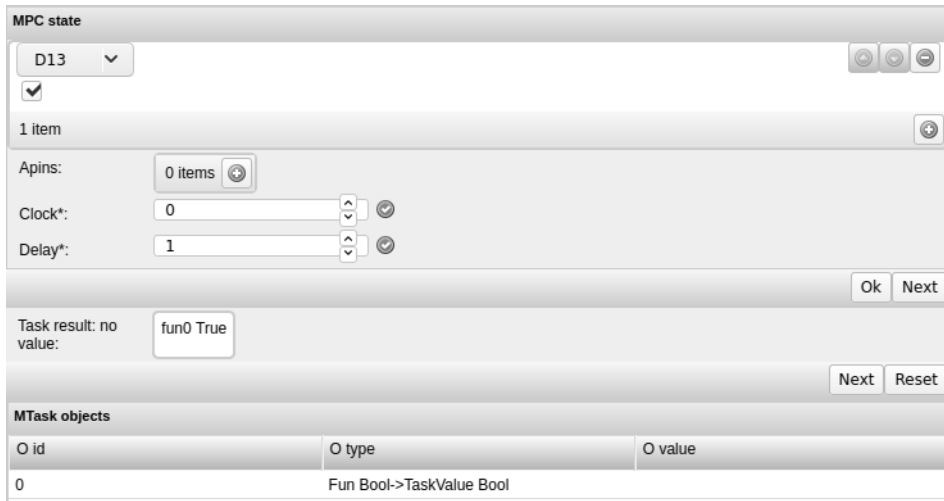
Figure 5.2: Simulator interface for a blink program written in mTask.

and iTask programs as well. The integration with iTask is explained thoroughly later in chapter 6.

The mTask language together with iTask is a heterogeneous DSL. I.e. some components—for example the RTS on the microcontroller that executes the tasks—is largely unaware of the other components in the system, and it is executed on a completely different architecture. The mTask language is a TOP language with basic tasks tailored for IoT edge devices (see section 5.4). It uses expressions based a simply-typed λ-calculus with support for some basic types, arithmetic operations, and function definitions.

5.6   Conclusion

This chapter gave an overview of the complete mTask DSL. The mTask language is a rich TOP language tailored for IoT edge devices. The language is implemented as a class-based shallow embedded DSL in the pure functional host language Clean. The language uses an enriched lambda calculus as a host language, providing additional language constructs for arithmetic expressions, conditionals, functions, but also non-interactive basic tasks, task combinators, peripheral support, and integration with iTask. Terms in the language are just interfaces and can be interpreted by one or more interpretations. When using the byte code compiler, terms in the mTask language are type checked at compile time but are constructed and compiled at run time. This facilitates tailor-making tasks for the current work requirements.

Chapter 6

The integration of mTask and iTask

This chapter shows the integration of mTask and iTask by discussing:

- an architectural overview of mTask applications;
- the interface for connecting devices;
- the interface for lifting mTask tasks to iTask tasks;
- the interface for lowering iTask SDSs to mTask SDSs;
- and a non-trivial home automation example application using all integration mechanisms;

The mTask system is a TOP DSL for edge devices. It is a multi-view DSL, there are multiple interpretations possible for a single mTask term. The main interpretation of mTask terms is the byte code compiler, `:: BCInterpret a`. When using this interpretation and a few integration functions, mTask tasks are fully integrated in iTask. They execute as regular iTask tasks and they can access SDSs from iTask. Devices in the mTask system are set up with a domain-specific OS and become little TOP engines in their own respect, being able to execute tasks.

Figure 6.1 shows the architectural layout of a typical IoT system created with iTask and mTask. The entire system is written as a single Clean specification where multiple tasks are executed at the same time. Tasks can access SDSs following the many-to-many communication pattern and multiple clients can work on the same task. The diagram contains three labelled arrows that denote the integration functions between iTask and mTask. Devices are connected to the system using the `withDevice` function (see section 6.1). There can be multiple devices connected to a single iTask host at the same time. Using `liftmTask`, mTask tasks are lifted to a device (see section 6.2). It is possible to execute multiple tasks on a single device. SDSs from iTask are lowered to the mTask device using `lowerSds` (see section 6.3).
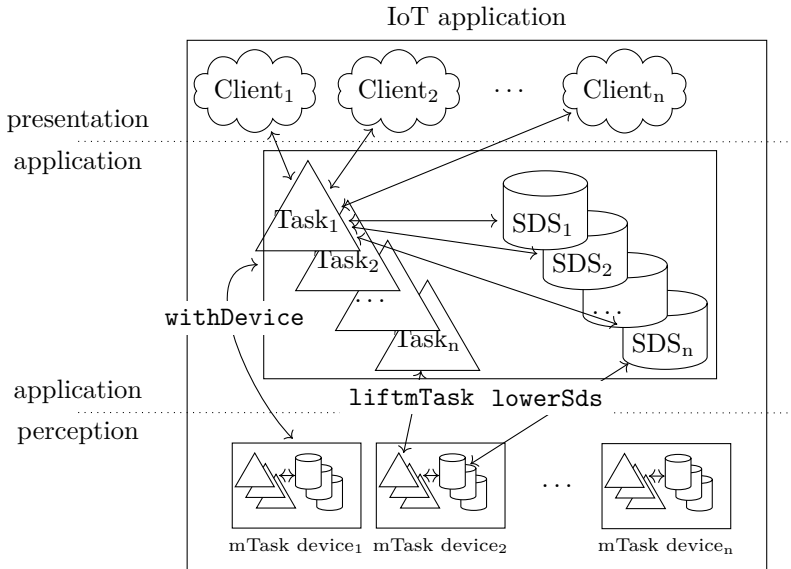
IoT application

Figure 6.1: An architectural overview of an iTask/mTask application.

6.1 Connecting edge devices

Edge devices in an mTask application are always coordinated by a server. This means that they wait for a server to take initiative, set up a connection, and send the work. The heavy lifting of connecting an mTask device to an iTask server is done with the `withDevice` iTask function. This function has two parameters, a communication specification, and a function using a device handle. The device handle is required to interact with mTask devices, e.g. lift tasks. By using HOAS like this, setting up and tearing down the connection to the device is fully controlled.

All communication with a device happens through a so-called *channels* SDS. The channels contain three fields, a queue of messages that are received, a queue of messages to send, and a stop flag. Every communication method that implements the `channelSync` class can provide the communication with an mTask device. At the time of writing, serial port, direct TCP, and MQTT over TCP are supported communication methods (see appendix B.1.8). Internally, the `withDevice` task sets up the communication, exchanges specifications with the device, executes the inner task while handling errors, and finally cleans up after closing. Listing 6.1 shows the types and interface for connecting devices.

```
:: MTDevice //abstract
:: Channels :== ([MTMessageFro], [MTMessageTo], Bool)
class channelSync a :: a (Shared sds Channels) → Task () | RWShared sds
withDevice :: a (MTDevice → Task b)
    → Task b | iTask b & channelSync, iTask a
```

Listing (Clean) 6.1: Device communication interface in mTask.

6.1.1 Implementation

Listing 6.2 shows a pseudocode implementation of the `withDevice` function. The `MTDevice` abstract type is internally represented as three iTask SDS that contain all the current information about the tasks. The first SDS is the information about the RTS of the device, i.e. metadata on the tasks that are executing, the hardware specification and capabilities, and a list of fresh task identifiers. The second SDS is a map storing downstream SDS updates. When a lowered SDS is updated on the device, a message is sent to the server. This message is initially queued in the map in order to properly handle multiple updates asynchronously. Finally, the `MTDevices` type contains the communication channels.

   The `withDevice` task itself first constructs the SDSs using the iTask function `withShared`. Then, it performs the following four tasks in parallel to monitor the edge device.

1. The channels are synchronised using the overloaded `channelSync` function. Errors that occur here are converted to the proper mTask or iTask exception.
2. The shutdown flag of the channels is watched. If the connection is lost with the device unexpectedly, an mTask exception is thrown.
3. The received messages in the channels are watched and processed. Depending on the type of message, either the device information SDS is updated, or the SDS update is added to the lowered SDS updates SDS.
4. A request for a specification is sent. Once the specification is received, the device task is run. The task value of this device task is then used as the task value of the `withDevice` task.

```
withDevice :: a (MTDevice → Task b) → Task b | ···
withDevice spec deviceTask =
    withShared default \dev→
    withShared newMap \sdsupdates→
    withShared ([], [MTTSpecRequest], False) \channels→
        parallel
            [ channelSync spec channels
            , watchForShutdown channels
            , watchChannelMessages dev channels
            , waitForSpecification
                >>| deviceTask (MTDevice dev sdsupdates channels)
                >>* [OnValue $ ifStable $ \_→issueShutdown]
            ]
```

Listing (Clean) 6.2: Pseudocode for the `withDevice` function in mTask.

If at any stage an unrecoverable device error occurs, an iTask exception is thrown in the `withDevice` task. This exception can be caught in order to devise fail-safe mechanisms. For example, if a device fails, the task can be sent to another device as can be seen in listing 6.3. This function executes an mTask task on a pool of devices connected through TCP. If a device error occurs during execution, the next device in the pool is tried until the pool is exhausted. If another type of error occurs, it is re-thrown for a parent task to catch.

```
failover :: [TCPSettings] (Main (MTask BCInterpret a)) → Task a
failover []      _     = throw "Exhausted device pool"
failover [d:ds] mtask = try (withDevice d (liftmTask mtask)) except
where except MTEUnexpectedDisconnect = failover ds mtask
      except e                       = throw e
```
Listing (Clean) 6.3: An mTask failover combinator.

6.2   Lifting mTask tasks

Once the connection with the device is established, mTask tasks are lifted to
iTask tasks using the `liftmTask` function (see listing 6.4). Given an mTask task
in the `BCInterpret` view and a device handle obtained from `withDevice`, an iTask
task is returned. This iTask task proxies the mTask task that is executed on the
microcontroller. So, when another task observes the task value, the actual task
value from the microcontroller is observed.

```
liftmTask :: (Main (MTask BCInterpret a)) MTDevice → Task a | iTask a
```
Listing (Clean) 6.4: The interface for lifting mTask tasks to iTask tasks.

6.2.1   Implementation

Listing 6.5 shows the pseudocode for the `liftmTask` implementation The first
argument is the task and the second argument is the device which is an ADT
containing the SDSs referring to the device information, the SDS update queue,
and the channels. First a fresh identifier for the task is generated using the device
state. With this identifier, the cleanup hook can be installed. This is done to
assure the task is removed from the edge device if the iTask task coordinating it is
destroyed. Tasks in iTask are destroyed when for example they are executed in
parallel with another task and the parallel combinator terminates, or when the
condition to step holds in a sequential task combination. Then the mTask compiler
is invoked, its only argument besides the task is a function doing something with
the results of the compilation, i.e. the lowered SDSs and the messages containing
the compiled and serialised task. With the result of the compilation, the task
can be executed. First the messages are put in the channels, sending them to the
device. Then, in parallel:

1. the value is watched by looking in the device state SDS, this task also
   determines the task value of the whole task;
2. the downstream SDSs are monitored, i.e. the `sdsupdates` SDS is monitored
   and updates from the device are applied to the associated iTask SDS;
3. the upstream SDSs are monitored by spawning tasks that watch these SDSs,
   if one is updated, the novel value is sent to the edge device.

Sending the complete byte code to the device is not always a suitable option.
For example, when the device is connected through an unstable or slow connection,

```
liftmTask :: (Main (MTask BCInterpret a)) MTDevice → Task a | iTask a
liftmTask task (MTDevice dev sdsupdates channels)
    = freshTaskId dev
    >>= \tid→withCleanupHook (sendmessage [MTTTaskDel tid] channels) (
        compile task \mrefs msgs→
                sendMessage msgs channels
            >>| waitForReturnAndValue tid dev
            -|| watchSharesDownstream mrefs tid sdsupdates
            -|| watchSharesUpstream mrefs channels tid)
```

Listing (Clean) 6.5: Pseudocode implementation for `liftmTask`.

sending the entire byte code induces a lot of delay. To mitigate this problem, mTask tasks can be preloaded on a device. Preloading means that the task is compiled and integrated into the device firmware. On receiving a `TaskPrep`, a hashed value of the task to be sent is included. The device then checks the preloaded task registry and uses the local preloaded version if the hash matches. Of course this only works for tasks that are not tailor-made for the current work specification and not depend on run time information. The interface for task preloading can be found in listing 6.6. Given an mTask task, a header file is created that should be placed in the source code directory of the RTS before building to include it in the firmware.

```
preloadTask :: (Main (MTask BCInterpret a)) → Task String
```

Listing (Clean) 6.6: Preloading tasks in mTask.

6.3   Lowering iTask shared data sources

Lowering iTask SDSs to mTask SDSs is something that mostly happens at the DSL level using the `lowerSds` function (see listing 6.7). Lowering an SDS proxies the iTask SDS for use in mTask. SDSs in mTask always have an initial value. For regular SDS this value is given in the source code, for lowered iTask SDSs this value is obtained by reading the values once just before sending the task to the edge device. On the device, there is just one difference between lowered SDSs and regular SDSs: after changing a lowered SDS, a message is sent to the server containing this new value. The `withDevice` task (see section 6.1) receives and processes this message by writing to the iTask SDS. Tasks watching this SDS get notified then through the normal notification mechanism of iTask. Section 7.2.5 shows the implementation of this type class for the byte code compiler.

```
class lowerSds v where
    lowerSds :: ((v (Sds t)) → In (Shared sds t) (Main (MTask v u)))
        → Main (MTask v u) | RWShared sds
```

Listing (Clean) 6.7: Lowered iTask SDSs in mTask.

As an example, listing 6.8 shows a light switch function producing an iTask/mTask task when given a device handle. First an iTask SDS of the type boolean

is created. This boolean represents the state of the light. The mTask task uses this SDS to turn on or off the light. The iTask task that runs in parallel allows interactive updating of this state.

```
lightswitch :: MTDevice → Task Bool
lightswitch dev  = withShared False \sh→
          liftmTask (mtask sh) dev
      -|| updateSharedInformation [] sh
      <<@ Hint "Light switch"
where
    mtask :: (Shared sds Bool) → Main (MTask v Bool)
        | mtask, lowerSds v & RWShared sds
    mtask sh =
        declarePin D13 PMOutput \ledPin→
        lowerSds \ls=sh
        In fun \f=(\st→
                getSds ls
            >>*. [IfValue (\v→v !=. st) (writeD ledPin)]
            >>=. f)
        In {main=getSds ls >>~. f}
```

Listing (Clean) 6.8: Interactive light switch program in mTask.

6.4 Conclusion

This chapter explained the integration of mTask programs with iTask. Using just three iTask functions, mTask devices are integrated in iTask seamlessly. Devices, using any supported type of connection, are integrated in iTask using the `withDevice` function. Once connected, mTask tasks are sent to the device for execution using `liftmTask`, lifting them to full-fledged iTask tasks. To lower the bandwidth, tasks can also be preloaded. Furthermore, the mTask tasks interact with iTask SDSs using the `lowerSds` construct. All of this together allows programming all layers of an IoT system from a single source and in a single paradigm. All details regarding interoperation are automatically taken care of. The following section contains an elaborate example using all integration functions that has deliberately been placed after the conclusion for formatting reasons.

let p = [*['This page would be intentionally blank if I were not telling you that ']*:p] **in** p

6.5   Home automation

This section presents an interactive home automation program (listing 6.9) to illustrate the dynamic integration of the mTask language and the iTask system. All layers of IoT systems are used in this application. The presentation layer consists of an automatically generated web interface for the user to control which tasks are sent to a device for execution. The application layer is the iTask server, the coordinator of the tasks in the system that also stores the data. The perception layer is populated by two devices: an Arduino UNO, and an ESP8266 based prototyping board called NodeMCU. Lines 1 to 2 show the specification for the devices. The UNO is connected via serial using the UNIX filepath `/dev/ttyACM0` and the default serial port settings. The NodeMCU is connected via TCP over Wi-Fi and hence the `TCPSettings` record is used.

The code is split up into an iTask part and several mTask parts. Lines 4 to 8 contains the iTask task that coordinates the IoT application. First the devices are connected (lines 5 to 6) followed by launching a `parallel` task, visualised as a tabbed window, and a shutdown button to terminate the program (lines 7 to 8). This parallel task is the controller of the tasks that run on the edge devices. It contains one task that allows adding new tasks (using `appendTask`) and all other tasks in the process list will be mTask tasks once they are added by the user. The controller task, `chooseTask` as shown in lines 10 to 19, allows the user to pick a task, and sending it to the specified device. Tasks are picked by index from the `tasks` list (lines 22 to 39) using `enterChoice`. The interface that is generated for this is seen in figure 6.2a. After selecting the task, a device is selected (see figure 6.2b and line 13). When both a task and a device are selected, an iTask task is added to the process list using `appendTask`. Using the helper function `mkTask`, the actual task is selected from the `tasks` list and executed by providing it the device argument.

The `tasks` list contains named mTask tasks that can be sent to the device. When selecting the `temperature` task, the current temperature is shown to the user (figure 6.2c). This task just sends a simple temperature monitoring task to the device using `liftmTask` and provides a view on its task value using the `>&>` iTask combinator. This combinator allows the observation of the left-hand side task's value through an SDS. The light switch task at lines 31 to 34 is a task that has bidirectional interaction using the definition of `lightswitch` shown in listing 6.8. Using `lowerSds`, the server-side status of the light switch is synchronised with the actual light attached to the GPIO pin. Finally, some tasks contain significant iTask portions as well. The remote computation task first queries the user for a number and then constructs a tailor-made task to send to the device to perform a computation, i.e. it calculates the factorial for the given number.

(a) Select task.   (b) Select device.   (c) View result.
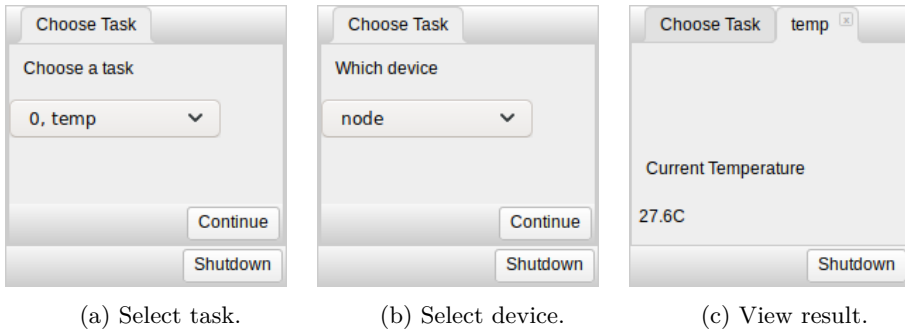
Figure 6.2: Screenshots of the home automation example program in action.

```
1   arduino = {TTYSettings | zero & devicePath="/dev/ttyACM0"}
2   nodeMCU = {TCPSettings | host="192.168.0.1", port=8123, pingTimeout= ?None}
3
4   autoHome :: Task ()
5   autoHome = withDevice arduino \dev1→
6             withDevice nodeMCU \dev2→
7      parallel [(Embedded, chooseTask dev1 dev2)] [] <<@ ArrangeWithTabs True
8      >>* [OnAction (Action "Shutdown") (always (shutDown 0))]
9
10  chooseTask :: MTDevice MTDevice (SharedTaskList ()) → Task ()
11  chooseTask dev1 dev2 stl = tune (Title "Run a task") $
12      enterChoice [] (zip2 [0..] (map fst tasks)) <<@ Hint "Choose a task"
13      >>? \(i, n)→enterChoice [] ["arduino", "node"]
14            <<@ Hint "Which device?"
15      >>? \device→appendTask Embedded (mkTask n i device) stl
16      >-| chooseTask dev1 dev2 stl
17  where
18      mkTask n i device stl = ((snd (tasks !! i) $ dev)
19          >>* [OnAction ActionClose $ always $ return ()]) <<@ Title n
20      where dev = if (device == "node") dev2 dev1
21
22  tasks :: [(String, MTDevice → Task ())]
23  tasks =
24      [ ("temp", \dev→
25          liftmTask (DHT (DHT_DHT (DigitalPin D6) DHT22) \dht→
26                  {main=temperature dht}
27              ) dev
28          >&> \t→viewSharedInformation
29                  [ViewAs \i→toString (fromMaybe 0.0 i) +++ "C"] t
30              <<@ Hint "Current Temperature" @! ())
31      , ("lightswitch", \dev→
32          withShared False \sh→
33                  liftmTask (lightswitch sh) dev
34              -|| updateSharedInformation [] sh <<@ Hint "Switch")
35      , ("remote computation", \dev→
36                  updateInformation [] 5 <<@ Hint "Factorial of what?"
37          >>? \i→liftmTask (factorial i) dev
38          >>- \r→viewInformation [] r <<@ Hint "Result" @! ())
39      , ···]
```

Listing (Clean) 6.9: An example of a home automation program.

Chapter 7

The implementation of mTask

This chapter shows the implementation of the mTask system by:

- showing the compilation and execution toolchain;
- showing the implementation of the byte code compiler for the mTask language;
- elaborating on the implementation and architecture of the RTS of mTask;
- and explaining the machinery used to automatically serialise and deserialise data to and fro the device.

The mTask system targets resource-constrained edge devices that have little memory, processor speed, and communication. Such edge devices are often powered by microcontrollers, tiny computers specifically designed for embedded applications. The microcontrollers usually have flash-based program memory which wears out fairly quickly. For example, the flash memory of the popular atmega328p powering the Arduino UNO is rated for *10 000* write cycles. While this sounds like a lot, if new tasks are sent to the device every minute or so, a lifetime of only seven days is guaranteed. Hence, for dynamic applications, storing the program in the RAM of the device and thus interpreting this code is necessary in order to save precious write cycles of the program memory. In the mTask system, the mTask RTS, a domain-specific OS, is responsible for interpreting the programs.

Programs in mTask are DSL terms constructed at run time in an iTask system. Figure 7.1 shows the compilation and execution toolchain of such programs. First, the source code is compiled to a byte code specification, this specification contains the compiled main expression, the functions, and the SDS and peripheral configuration. How an mTask task is compiled to this specification is shown in section 7.1.
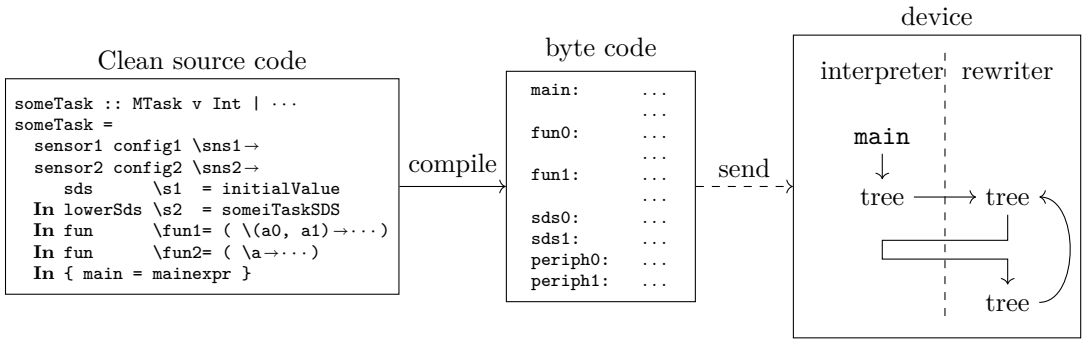
Figure 7.1: Compilation and execution toolchain of mTask programs.

This package is then sent to the RTS of the device for execution. In order to execute a task, first the main expression is evaluated in the interpreter, resulting in a task tree. Then, using small-step reduction, the task tree is continuously rewritten by the rewrite engine of the RTS. At times, the reduction requires the evaluation of expressions, using the interpreter. During every rewrite step, a task value is produced. On the device, the RTS may have multiple tasks at the same time active. By interleaving the rewrite steps, parallel operation is achieved. The design, architecture and implementation of the RTS is shown in section 7.3.

7.1 Compiler

The byte code compiler for mTask is an interpretation of the mTask language. In order to compile terms, instances for all mTask type classes are required for the :: `BCInterpret a` type. Terms in mTask are constructed and compiled at run time, but type checked at compile time in the host language Clean. The compiled tasks are sent to the device for interpretation. The result of the compilation is the byte code and some metadata regarding the used peripherals and SDSs. The compilation target is the interpreter of the mTask RTS. In order to keep the hardware requirements down, all expressions are evaluated on a stack. Rewriting of tasks uses the same stack and also a heap. The heap usage is minimised by applying aggressive memory management. A detailed overview of the RTS including the interpreter and rewriter is found in section 7.3.

7.1.1 Compiler infrastructure

The byte code compiler interpretation for the mTask language is implemented as a monad stack containing a writer monad and a state monad. The writer monad is used to generate code snippets locally without having to store them in the monadic values. The state monad accumulates the code, and stores the state the compiler requires. Listing 7.1 shows the data type for the state, storing: function the compiler currently is in; code of the main expression; context (see section 7.2.4);

code for the functions; next fresh label; a list of all the used SDSs, either local SDSs containing the initial value (`Left`) or lowered SDSs (see section 6.3) containing a reference to the associated iTask SDS; and finally there is a list of peripherals used.

```
:: BCInterpret a :== StateT BCState (WriterT [BCInstr] Identity) a
:: BCState =
    { bcs_infun        :: JumpLabel
    , bcs_mainexpr     :: [BCInstr]
    , bcs_context      :: [BCInstr]
    , bcs_functions    :: Map JumpLabel BCFunction
    , bcs_freshlabel   :: JumpLabel
    , bcs_sdses        :: [Either String255 MTLens]
    , bcs_hardware     :: [BCPeripheral]
    }
:: BCFunction =
    { bcf_instructions :: [BCInstr]
    , bcf_argwidth     :: UInt8
    , bcf_returnwidth  :: UInt8
    }
```
Listing (Clean) 7.1: The type for the mTask byte code compiler.

Executing the compiler is done by providing an initial state and running the monad. After compilation, several post-processing steps are applied to make the code suitable for the microprocessor. First, in all tail call `BCReturn` instructions are replaced by `BCTailCall` instructions to optimise the tail calls. Furthermore, all byte code is concatenated, resulting in one big program. Many instructions have commonly used arguments, so shorthands are introduced to reduce the program size. For example, the `BCArg` instruction is often called with argument *0* to *2* and can be replaced by the `BCArg0` to `BCArg2` shorthands. Furthermore, redundant instructions such as pop directly after push are removed as well in order not to burden the code generation with these intricacies. Finally, the labels are resolved to represent actual program addresses instead of the freshly generated identifiers. After the byte code is ready, the lowered SDSs are resolved to provide an initial value for them. The byte code, SDS specification and peripheral specifications are the result of the process, ready to be sent to the device.

7.1.2   Instruction set

The instruction set is a fairly standard stack machine instruction set extended with special TOP instructions for creating task tree nodes. All instructions are housed in a Clean ADT and serialised to the byte representation using generic functions (see section 7.4). Type synonyms and newtypes are used to provide insight on the arguments of the instructions (listing 7.2). Labels are always two bytes long, all other arguments are one byte long.

```
:: ArgWidth    :== UInt8        :: ReturnWidth :== UInt8
:: Depth       :== UInt8        :: Num         :== UInt8
:: SdsId       :== UInt8        :: JumpLabel   =: JL UInt16
```
Listing (Clean) 7.2: Type synonyms for instructions arguments.

Listing 7.3 shows an excerpt of the Clean type that represents the instruction set. Shorthand instructions such as instructions with inlined arguments are omitted for brevity. Detailed semantics for the instructions are given in appendix C. One notable instruction is the `MkTask` instruction, it allocates and initialises a task tree node and pushes a pointer to it on the stack.

```
:: BCInstr
    //Jumps
    = BCJumpF JumpLabel | BCLabel JumpLabel | BCJumpSR ArgWidth JumpLabel
    | BCReturn ReturnWidth ArgWidth
    | BCTailcall ArgWidth ArgWidth JumpLabel
    //Arguments
    | BCArgs ArgWidth ArgWidth
    //Task node creation and refinement
    | BCMkTask BCTaskType | BCTuneRateMs | BCTuneRateSec
    //Stack ops
    | BCPush String255 | BCPop Num | BCRot Depth Num | BCDup | BCPushPtrs
    //Casting
    | BCItoR | BCItoL | BCRtoI | ···
    // arith
    | BCAddI | BCSubI | ···
    ...

:: BCTaskType
    = BCStableNode ArgWidth | BCUnstableNode ArgWidth
    // Pin io
    | BCReadD | BCWriteD | BCReadA | BCWriteA | BCPinMode
    // Interrupts
    | BCInterrupt
    // Repeat
    | BCRepeat
    // Delay
    | BCDelay | BCDelayUntil
    // Parallel
    | BCTAnd | BCTOr
    //Step
    | BCStep ArgWidth JumpLabel
    //Sds ops
    | BCSdsGet SdsId | BCSdsSet SdsId | BCSdsUpd SdsId JumpLabel
    // Rate limiter
    | BCRateLimit
    ////Peripherals
    //DHT
    | BCDHTTemp UInt8 | BCDHTHumid UInt8
    ...
```

Listing (Clean) 7.3: The type housing the instruction set in mTask.

Table 7.1: An overview of the compilation rules.

Scheme	Description
$\mathcal{E}[\![e]\!]\ r$	Generates code for expressions given the context r
$\mathcal{F}[\![e]\!]$	Generates the code for functions.
$\mathcal{S}[\![e]\!]\ r\ w$	Generates the code for the step continuations given the context r and the width w of the left-hand side task value.

7.2 Compilation rules

This section describes the compilation rules, the translation from AST to byte code. The compilation scheme consists of three schemes/functions. Double vertical bars, e.g. $\|\ a_i\ \|$, denote the number of stack cells required to store the argument.

Some schemes have a context r as an argument which contains information about the location of the arguments in scope. More information is given in the schemes requiring such arguments.

7.2.1 Expressions

Almost all expression constructions are compiled using $\mathcal{E}$. The argument of $\mathcal{E}$ is the context (see section 7.2.2). Values are always placed on the stack; tuples and other compound data types are unpacked. Function calls, function arguments and tasks are also compiled using $\mathcal{E}$ but their compilations is explained later.

$$\mathcal{E}[\![\texttt{lit}\ e]\!]\ r = \texttt{BCPush (bytecode e)};$$

$$\mathcal{E}[\![e_1\ \texttt{+.}\ e_2]\!]\ r = \mathcal{E}[\![e_1]\!]\ r; \mathcal{E}[\![e_2]\!]\ r; \texttt{BCAdd};$$

$$\textit{Similar for other binary operators}$$

$$\mathcal{E}[\![\texttt{Not}\ e]\!]\ r = \mathcal{E}[\![e]\!]\ r; \texttt{BCNot};$$

$$\textit{Similar for other unary operators}$$

$$\mathcal{E}[\![\texttt{If}\ e_1\ e_2\ e_3]\!]\ r = \mathcal{E}[\![e_1]\!]\ r; \texttt{BCJmpF}\ l_{else}; \quad \mathcal{E}[\![e_2]\!]\ r; \texttt{BCJmp}\ l_{endif};$$

$$\texttt{BCLabel}\ l_{else}; \mathcal{E}[\![e_3]\!]\ r; \quad \texttt{BCLabel}\ l_{endif};$$

$$\textit{Where } l_{else} \textit{ and } l_{endif} \textit{ are fresh labels}$$

$$\mathcal{E}[\![\texttt{tupl}\ e_1\ e_2]\!]\ r = \mathcal{E}[\![e_1]\!]\ r; \mathcal{E}[\![e_2]\!]\ r;$$

$$\textit{Similar for other unboxed compound data types}$$

$$\mathcal{E}[\![\texttt{first}\ e]\!]\ r = \mathcal{E}[\![e]\!]\ r; \texttt{BCPop}\ w;$$

$$\textit{Where w is the width of the right value and}$$

$$\textit{similar for other unboxed compound data types}$$

$$\mathcal{E}[\![\texttt{second}\ e]\!]\ r = \mathcal{E}[\![e]\!]\ r; \texttt{BCRot}\ (w_l + w_r)\ w_r; \texttt{BCPop}\ w_l;$$

$$\textit{Where } w_l \textit{ is the width of the left and, } w_r \textit{ of the right value}$$

$$\textit{similar for other unboxed compound data types}$$

Translating $\mathcal{E}$ to Clean code is very straightforward, it basically means writing the instructions to the writer monad. Almost always, the type of the interpretation is not used, i.e. it is a phantom type. To still have the functions return the correct type, the `tell`` [1] helper is used. This function is similar to the writer monad's `tell` function but is cast to the correct type. Listing 7.4 shows the implementation for the arithmetic and conditional expressions. Note that r, the context, is not an explicit argument here but stored in the state.

```
instance expr BCInterpret where
    lit   t   = tell` [BCPush (toByteCode{|*|} t)]
    (+.)  a b = a >>| b >>| tell` [BCAdd]
    ...
    If c t e = freshlabel >>= \elselabel→freshlabel >>= \endiflabel→
        c >>| tell` [BCJumpF elselabel] >>|
        t >>| tell` [BCJump endiflabel,BCLabel elselabel] >>|
        e >>| tell` [BCLabel endiflabel]
```

Listing (Clean) 7.4: Interpretation implementation for the arithmetic and conditional functions.

7.2.2   Functions

Compiling functions and other top-level definitions is done using in $\mathcal{F}$, which generates bytecode for the complete program by iterating over the functions and ending with the main expression. When compiling the body of the function, the arguments of the function are added to the context so that the addresses can be determined when referencing arguments. The main expression is a special case of $\mathcal{F}$ since it neither has arguments nor something to continue. Therefore, it is just compiled using $\mathcal{E}$ with an empty context.

$$\mathcal{F}[\![main = m]\!] = \mathcal{E}[\![m]\!] \; [];$$
$$\mathcal{F}[\![f \; a_0 \ldots a_n = b \; \textbf{In} \; m]\!] = \texttt{BCLabel} \; f; \mathcal{E}[\![b]\!] \; [\langle f, i \rangle, i \in \{(\Sigma_{i=0}^n \; \| \; a_i \; \|)..0\}];$$
$$\texttt{BCReturn} \; \| \; b \; \| \; n; \mathcal{F}[\![m]\!];$$

A function call starts by pushing the stack and frame pointer, and making space for the program counter (figure 7.2a) followed by evaluating the arguments in reverse order (figure 7.2b). On executing `BCJumpSR`, the program counter is set, and the interpreter jumps to the function (figure 7.2c). When the function returns, the return value overwrites the old pointers and the arguments. This occurs right after a `BCReturn` (figure 7.2d). Putting the arguments on top of pointers and not reserving space for the return value uses little space and facilitates tail call optimisation.

---

[1] `tell`` :: [BCInstr] → BCInterpret a`

	...		...		...		...
	fp_{old}		fp_{old}		fp_{old}		ret_0
	sp_{old}		sp_{old}		sp_{old}		$ret_{...}$
	0		0		pc_{old}		ret_n
			arg_n		arg_n		
			$arg_{...}$		$arg_{...}$		
			arg_0		arg_0		
	...		...		...		...

(a) `BCPushPtrs`.   (b) Arguments.   (c) `BCJumpSR`.   (d) `BCReturn`.
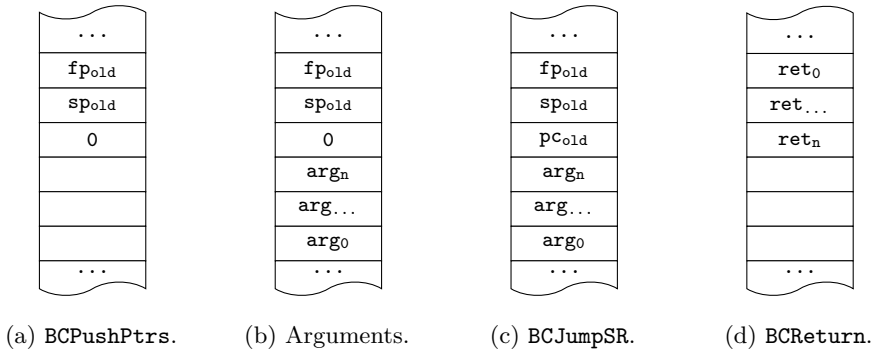
Figure 7.2: The stack layout during function calls.

Calling a function and referencing function arguments are an extension to $\mathcal{E}$ as shown below. Arguments may be at different places on the stack at different times (see section 7.2.4) and therefore the exact location is always is determined from the context using `findarg`.[2] Compiling argument a_{fi}, the ith argument in function f, consists of traversing all positions in the current context. Arguments wider than one stack cell are fetched in reverse to reconstruct the original order.

$$\mathcal{E}[\![f(a_0,\ldots,a_n)]\!] \ r = \texttt{BCPushPtrs}; \mathcal{E}[\![a_i]\!] \ r \text{ for all } i \in \{n\ldots 0\}; \texttt{BCJumpSR} \ n \ f;$$
$$\mathcal{E}[\![a_{fi}]\!] \ r = \texttt{BCArg} \ \text{findarg}(r,f,i) \text{ for all } i \in \{w\ldots v\};$$
$$v = \Sigma_{j=0}^{i-1} \parallel a_{fj} \parallel \quad \text{and} \quad w = v+ \parallel a_{fi} \parallel$$

Translating the compilation schemes for functions to Clean is not as straightforward as other schemes due to the nature of shallow embedding in combination with the use of state. The `fun` class has a single function with a single argument. This argument is a Clean function that—when given a callable Clean function representing the mTask function—produces the `main` expression and a callable function. To compile this, the argument must be called with a function representing a function call in mTask. Listing 7.5 shows the implementation for this as Clean code. To uniquely identify the function, a fresh label is generated. The function is then called with the `callFunction` helper function that generates the instructions that correspond to calling the function. That is, it pushes the pointers, compiles the arguments, and writes the `JumpSR` instruction. The resulting structure (`g In m`) contains a function representing the mTask function (`g`) and the `main` structure to continue with. To get the actual function, `g` must be called with representations for the argument, i.e. using `findarg` for all arguments. The arguments are added to the context using `infun` and `liftFunction` is called with the label, the argument width and the compiler. This function executes the compiler, decorates the instructions with a label and places them in the function dictionary together with the metadata

---

[2]`findarg [l`:r] l = if (l == l`) 0 (1 + findarg r l)`

such as the argument width. After lifting the function, the context is cleared again
and compilation continues with the rest of the program.

```Clean
instance fun (BCInterpret a) BCInterpret | type a where
    fun def = {main=freshlabel >>= \funlabel→
        let (g In m) = def \a→callFunction funlabel (toByteWidth a) [a]
            argwidth = toByteWidth (argOf g)
        in  addToCtx funlabel zero argwidth
        >>| infun funlabel
            (liftFunction funlabel argwidth
                (g (retrieveArgs funlabel zero argwidth)
                ) ?None)
        >>| clearCtx >>| m.main
        }

argOf :: ((m a) → b) a → UInt8 | toByteWidth a
callFunction :: JumpLabel UInt8 [BCInterpret b] → BCInterpret c | ⋯
liftFunction :: JumpLabel UInt8 (BCInterpret a) (?UInt8) → BCInterpret ()
infun :: JumpLabel (BCInterpret a) → BCInterpret a
```

Listing (Clean) 7.5: The interpretation implementation for functions.

7.2.3   Tasks

Task trees are created with the `BCMkTask` instruction that allocates a node and
pushes a pointer to it on the stack. It pops arguments from the stack according to
the given task type. The following extension of $\mathcal{E}$ shows this compilation scheme
(except for the step combinator, explained in section 7.2.4).

$$\mathcal{E}[\![\mathtt{rtrn}\ e]\!]\ r = \mathcal{E}[\![e]\!]\ r; \mathtt{BCMkTask\ BCStable}_{\|e\|};$$
$$\mathcal{E}[\![\mathtt{unstable}\ e]\!]\ r = \mathcal{E}[\![e]\!]\ r; \mathtt{BCMkTask\ BCUnstable}_{\|e\|};$$
$$\mathcal{E}[\![\mathtt{readA}\ e]\!]\ r = \mathcal{E}[\![e]\!]\ r; \mathtt{BCMkTask\ BCReadA};$$
$$\mathcal{E}[\![\mathtt{writeA}\ e_1\ e_2]\!]\ r = \mathcal{E}[\![e_1]\!]\ r; \mathcal{E}[\![e_2]\!]\ r; \mathtt{BCMkTask\ BCWriteA};$$
$$\mathcal{E}[\![\mathtt{readD}\ e]\!]\ r = \mathcal{E}[\![e]\!]\ r; \mathtt{BCMkTask\ BCReadD};$$
$$\mathcal{E}[\![\mathtt{writeD}\ e_1\ e_2]\!]\ r = \mathcal{E}[\![e_1]\!]\ r; \mathcal{E}[\![e_2]\!]\ r; \mathtt{BCMkTask\ BCWriteD};$$
$$\mathcal{E}[\![\mathtt{delay}\ e]\!]\ r = \mathcal{E}[\![e]\!]\ r; \mathtt{BCMkTask\ BCDelay};$$
$$\mathcal{E}[\![\mathtt{rpeat}\ e]\!]\ r = \mathcal{E}[\![e]\!]\ r; \mathtt{BCMkTask\ BCRepeat};$$
$$\mathcal{E}[\![e_1\ \mathtt{.||.}\ e_2]\!]\ r = \mathcal{E}[\![e_1]\!]\ r; \mathcal{E}[\![e_2]\!]\ r; \mathtt{BCMkTask\ BCOr};$$
$$\mathcal{E}[\![e_1\ \mathtt{.\&\&.}\ e_2]\!]\ r = \mathcal{E}[\![e_1]\!]\ r; \mathcal{E}[\![e_2]\!]\ r; \mathtt{BCMkTask\ BCAnd};$$

This compilation scheme translates to Clean code by first writing the arguments
and then the correct `BCMkTask` instruction. This is shown for the `.&&.` task in
listing 7.6.

```
instance .&&. BCInterpret where
    (.&&.) l r = l >>| r >>| tell` [BCMkTask BCTAnd]
```

Listing (Clean) 7.6: The byte code interpretation implementation for `rtrn`.

7.2.4   Sequential combinator

The `step` construct is a special type of task because the task value of the left-hand side changes over time. Therefore, the task continuations on the right-hand side are *observing* this task value and acting upon it. In the compilation scheme, all continuations are first converted to a single function that has two arguments: the stability of the task and its value. This function either returns a pointer to a task tree or fails (denoted by $\perp$). It is special because in the generated function, the task value of a task is inspected. Furthermore, it is a lazy node in the task tree: the right-hand side may yield a new task tree after several rewrite steps, i.e. it is allowed to create infinite task trees using step combinators. The function is generated using the $\mathcal{S}$ scheme that requires two arguments: the context r and the width of the left-hand side so that it can determine the position of the stability which is added as an argument to the function. The resulting function is basically a list of if-then-else constructions to check all predicates one by one. Some optimisation is possible here but has currently not been implemented.

$$\mathcal{E}[\![t_1 \texttt{>>*.}\, e_2]\!]\ r = \mathcal{E}[\![a_{f^i}]\!]\ r, \langle f, i \rangle \in r; \texttt{BCMkTask BCStable}_{\|r\|}; \mathcal{E}[\![t_1]\!]\ r;$$
$$\texttt{BCMkTask BCAnd}; \texttt{BCMkTask}\ (\texttt{BCStep}\ (\mathcal{S}[\![e_2]\!]\ (r + [\langle l_s, i \rangle])\ \|\ t_1\ \|));$$

$$\mathcal{S}[\![[]]\!]\ r\ w = \texttt{BCPush}\ \perp;$$
$$\mathcal{S}[\![\texttt{IfValue}\ f\ t : cs]\!]\ r\ w = \texttt{BCArg}(\|\ r\ \|+w); \texttt{BCIsNoValue};$$
$$\mathcal{E}[\![f]\!]\ r; \texttt{BCAnd};$$
$$\texttt{BCJmpF}\ l_1;$$
$$\mathcal{E}[\![t]\!]\ r; \texttt{BCJmp}\ l_2;$$
$$\texttt{BCLabel}\ l_1; \mathcal{S}[\![cs]\!]\ r\ w;$$
$$\texttt{BCLabel}\ l_2;$$
$$\textit{Where }l_1\textit{ and }l_2\textit{ are fresh labels}$$
$$\textit{Similar for } \texttt{IfStable}\ \textit{and}\ \texttt{IfUnstable}$$

The step combinator has a task as the left-hand side and a list of continuations at the right-hand side. First the context is evaluated ($\mathcal{E}[\![a_{f^i}]\!]\ r$). The context contains arguments from functions and steps that need to be preserved after rewriting. The evaluated context is combined with the left-hand side task value by means of a `.&&.` combinator to store it in the task tree so that it is available after a rewrite step.

(a) Without the embedded context.   (b) With the embedded context.
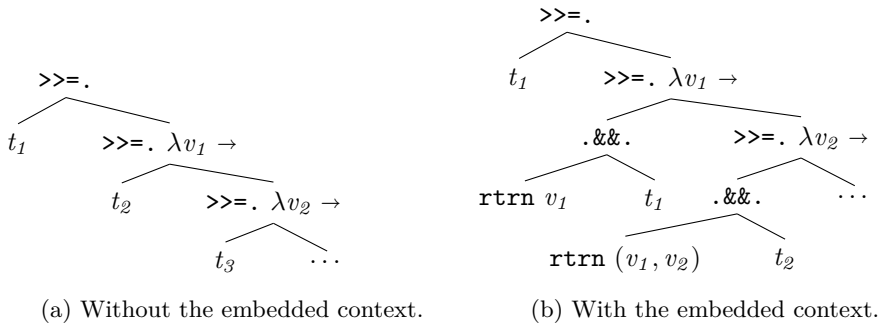
Figure 7.3: Context embedded in a virtual task tree.

This means that the task tree is transformed as seen in figure 7.3. In this figure, the expression `t1 >>=. \v1→t2 >>=. \v2→···` is shown.[3] Then, the right-hand side list of continuations is converted to an expression using $\mathcal{S}$.

The translation to Clean is given in listing 7.7.

```
instance step BCInterpret where
    (>>*.) lhs cont
        //Fetch a fresh label and fetch the context
        =   freshlabel >>= \funlab→gets (\s→s.bcs_context)
        //Generate code for lhs
        >>= \ctx→lhs
        //Possibly add the context
        >>| tell` (if (ctx =: []) []
                //The context is just the arguments up till now in reverse
                (   [BCArg (UInt8 i)\\i<-reverse (indexList ctx)]
                ++ map BCMkTask (bcstable (UInt8 (length ctx)))
                ++ [BCMkTask BCTAnd]
                ))
        //Increase the context
        >>| addToCtx funlab zero lhswidth
        //Lift the step function
        >>| liftFunction funlab
                //Width of the arguments is the width of the lhs plus the
                //stability plus the context
                (one + lhswidth + (UInt8 (length ctx)))
                //Body      label ctx width              continuations
                (contfun funlab (UInt8 (length ctx)))
                //Return width (always 1, a task pointer)
                (Just one)
        >>| modify (\s→{s & bcs_context=ctx})
        >>| tell` [BCMkTask (instr rhswidth funlab)]
```

---

[3]`t >>=. e` is a shorthand combinator for `t >>* [OnStable (\_→true) e]`.

```
toContFun :: JumpLabel UInt8 → BCInterpret a
toContFun steplabel contextwidth
    = foldr tcf (tell` [BCPush fail]) cont
where
    tcf (IfStable f t)
        = If ((stability >>| tell` [BCIsStable]) &. f val)
            (t val >>| tell` [])
    ...
    stability = tell` [BCArg (lhswidth + contextwidth)]
    val = retrieveArgs steplabel zero lhswidth
```

Listing (Clean) 7.7: Byte code compilation interpretation implementation for the step class.

7.2.5 Shared data sources

The compilation scheme for SDS definitions is a trivial extension to $\mathcal{F}$. While there is no code generated in the definition, the byte code compiler is storing all SDS data in the `bcs_sdses` field in the compilation state. Regular SDSs are stored as `Right String255` values. The SDSs are typed as functions in the host language, so an argument for this function must be created that represents the SDS on evaluation. For this, an `BCInterpret` is created that emits this identifier. When passing it to the function, the initial value of the SDS is returned. In the case of a local SDS, this initial value is stored as a byte code encoded value in the state and the compiler continues with the rest of the program. The SDS access tasks have a compilation scheme similar to other tasks (see section 7.2.3). The `getSds` task just pushes a task tree node with the SDS identifier embedded. The `setSds` task evaluates the value, lifts that value to a task tree node and creates an SDS set node.

$$\mathcal{F}[\![\text{sds } x = i \text{ In } m]\!] = \mathcal{F}[\![m]\!];$$

$$\mathcal{E}[\![\text{getSds } s]\!] \ r = \texttt{BCMkTask (BCSdsGet } s);$$
$$\mathcal{E}[\![\text{setSds } s \ e]\!] \ r = \mathcal{E}[\![e]\!] \ r; \texttt{BCMkTask BCStable}_{\|e\|};$$
$$\texttt{BCMkTask (BCSdsSet } s);$$

Listing 7.8 shows the implementation of the `sds` type class. First, the initial SDS value is extracted from the expression by bootstrapping the fixed point with a dummy value. This is safe because the expression on the right-hand side of the **In** is never evaluated. Then, using `addSdsIfNotExist`, the identifier for this particular SDS is either retrieved from the compiler state or generated freshly. This identifier is then used to provide a reference to the **def** definition to evaluate the main expression. Compiling `getSds` is a matter of executing the `BCInterpret` representing the SDS, which yields the identifier that can be embedded in the

instruction. Setting the SDS is similar: the identifier is retrieved, and the value is written to put in a task tree so that the resulting task can remember the value it has written.

```
:: Sds a = Sds Int
instance sds BCInterpret where
    sds def = {main =
            let (t In e) = def (abort "sds: expression too strict")
            in addSdsIfNotExist (Left $ String255 (toByteCode{|*|} t))
                >>= \sdsi→ let (t In e) = def (pure (Sds sdsi))
                    in e.main
        }
    getSds f   = f >>= \(Sds i)→ tell` [BCMkTask (BCSdsGet (fromInt i))]
    setSds f v = f >>= \(Sds i)→v >>| tell`
        (  map BCMkTask (bcstable (byteWidth v))
        ++ [BCMkTask (BCSdsSet (fromInt i))])
```
Listing (Clean) 7.8: Backend implementation for the SDS classes.

Lowered SDSs are stored in the compiler state as `Right MTLens` values. The compilation of the code and the serialisation of the data throws away all typing information. The `MTLens` is a type synonym for an SDS that represents the typeless serialised value of the underlying SDS. This is done so that the `withDevice` task can write the received SDS updates to the according SDS while the SDS is not in scope. The iTask notification mechanism then takes care of the rest. Such an SDS is created by using the `mapReadWriteError` which, given a pair of read and write functions with error handling, produces an SDS with the lens embedded. The read function transforms the typed value to a typeless serialised value. The write function will, given a new serialised value and the old typed value, produce a new typed value. It tries to decode the serialised value, if that succeeds, it is written to the underlying SDS, otherwise, an error is thrown otherwise. Listing 7.9 shows the implementation for this.

```
lens :: (Shared sds a) → MTLens | type a & RWShared sds
lens sds = mapReadWriteError
    ( \r→   Ok (fromString (toByteCode{|*|} r)
    , \w r→ ?Just <$> iTasksDecode (toString w)
    ) ?None sds
```
Listing (Clean) 7.9: Lens applied to lowered iTask SDSs in mTask.

Listing 7.10 shows the code for the implementation of `lowerSds` that uses the `lens` function shown earlier. It is very similar to the `sds` constructor in listing 7.8, only now a `Right` value is inserted in the SDS administration.

```
instance lowerSds BCInterpret where
    lowerSds def = {main =
            let (t In _) = def (abort "lowerSds: expression too strict")
            in addSdsIfNotExist (Right $ lens t)
                >>= \sdsi→let (_ In e) = def (pure (Sds sdsi)) in e.main
        }
```
Listing (Clean) 7.10: The implementation for lowering SDSs in mTask.

7.3   Run-time system

The RTS is a customisable domain-specific OS that takes care of the execution of tasks. Furthermore, it also takes care of low-level mechanisms such as the communication, multitasking, and memory management. Once a device is programmed with the mTask RTS, it can continuously receive new tasks without the need for reprogramming. The OS is written in portable C/C++ and only contains a small device-specific portion. In order to keep the abstraction level high and the hardware requirements low, much of the high-level functionality of the mTask language is implemented not in terms of lower-level constructs from mTask language but in terms of C/C++ code.

Most microcontroller software consists of a cyclic executive instead of an OS. This one loop function is continuously executed and all work is performed there. In the RTS of the mTask system, there is also such an event loop function. It is a function with a relatively short execution time that gets called repeatedly. The event loop consists of three distinct phases. After doing the three phases, the devices goes to sleep for as long as possible (see chapter 8 for more details on task scheduling).

7.3.1   Communication phase

In the first phase, the communication channels are processed. The exact communication method is a customisable device-specific option baked into the RTS. The interface is kept deliberately simple and consists of two layers: a link interface and a communication interface. Besides opening, closing and cleaning up, the link interface has three functions that are shown in listing 7.11. Consequently, implementing this link interface is very simple, but it is still possible to implement more advanced link features such as buffering. There are implementations for this interface for serial or Wi-Fi connections using Arduino, and TCP connections for Linux.

```cpp
bool    link_input_available(void);
uint8_t link_read_byte(void);
void    link_write_byte(uint8_t b);
```
Listing (C++) 7.11: Link interface of the mTask RTS.

The communication interface abstracts away from this link interface and is typed instead. It contains only two functions as seen in listing 7.12. There are implementations for direct communication, or communication using an MQTT broker. Both use the automatic serialisation and deserialisation shown in section 7.4.

```cpp
struct MTMessageTo receive_message(void);
void send_message(struct MTMessageFro msg);
```
Listing (C++) 7.12: Communication interface of the mTask RTS.

Processing the received messages from the communication channels happens synchronously and the channels are exhausted completely before moving on to the

next phase. There are several possible messages that can be received from the
server:

SpecRequest is a message instructing the device to send its specification. It is
received immediately after connecting. The RTS responds with a `Spec` answer
containing the specification.

TaskPrep tells the device a task is on its way. Especially on faster connections, it
may be the case that the communication buffers overflow because a big message
is sent while the RTS is busy executing tasks. This message allows the RTS to
postpone execution for a while, until the larger task has been received. The
server sends the task only after the device acknowledged the preparation by
sending a `TaskPrepAck` message.

Task contains a new task, its peripheral configuration, the SDSs, and the byte
code. The new task is immediately copied to the task storage but is only
initialised during the next phase. The device acknowledges the task by sending
a `TaskAck` message.

SdsUpdate notifies the device of the new value for a lowered SDS. The old value
of the lowered SDS is immediately replaced with the new one. There is no
acknowledgement required.

TaskDel instructs the device to delete a running task. Tasks are automatically
deleted when they become stable. However, a task may also be deleted when
the surrounding task on the server is deleted, for example when the task is on
the left-hand side of a step combinator and the condition to step holds. The
device acknowledges the deletion by sending a `TaskDelAck`.

Shutdown tells the device to reset.

7.3.2   Execution phase

The second phase performs one execution step for all tasks that wish for it. Tasks
are placed in a priority queue orderd by the time a task needs to execute. The RTS
selects all tasks that can be scheduled, see section 8.3 for more details. Execution
of a task is always an interplay between the interpreter and the rewriter. The
rewriter scans the current task tree and tries to rewrite it using small-step reduction.
Expressions in the tree are always strictly evaluated by the interpreter.

When a new task is received, the main expression is evaluated to produce a
task tree. A task tree is a tree structure in which each node represents a task
combinator and the leaves are basic tasks. If a task is not initialised yet, i.e. the
pointer to the current task tree is still null, the byte code of the main function
is interpreted. The main expression of mTask programs sent to the device before
execution always produces a task tree. Execution of a task consists of continuously
rewriting the task until its value is stable.

Rewriting is a destructive process, i.e. the rewriting is done in place. The
rewriting engine uses the interpreter when needed, e.g. to calculate the step contin-
uations. The rewriter and the interpreter use the same stack to store intermediate
values. Rewriting steps are small so that interleaving results in seemingly parallel
execution. In this phase new task tree nodes may be allocated. Both rewriting and
initialization are atomic operations in the sense that no processing on SDSs is done

(a) Initial task tree

(b) Task tree after the delay passed
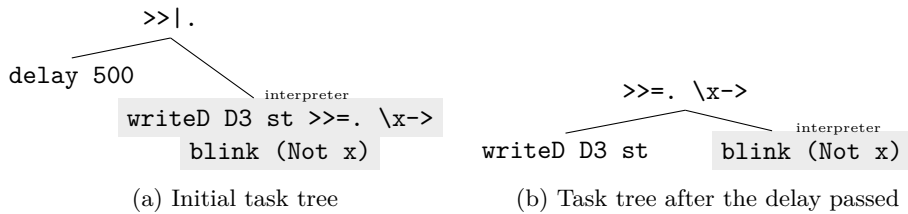
Figure 7.4: The task trees during reduction for a blink task in mTask.

other than SDS operations from the task itself. The host is notified if a task value is changed after a rewrite step by sending a `TaskReturn` message.

Take for example a blink task for which the code is shown in listing 7.13.

```
declarePin D13 PMOutput \ledPin→
fun \blink=(\st→delay (lit 500) >>|. writeD ledPin st >>=. blink o Not)
In {main = blink true}
```
Listing (Clean) 7.13: Code for a blink program.

On receiving this task, the task tree is still null and the initial expression `blink true` is evaluated by the interpreter. This results in the task tree shown in figure 7.4a. Rewriting always starts at the top of the tree and traverses to the leaves, the basic tasks that do the actual work. The first basic task encountered is the `delay` task, that yields no value until the time, *500* ms in this case, has passed. When the `delay` task yielded a stable value after a number of rewrites, the task continues with the right-hand side of the `>>|.` combinator by evaluating the expression (see figure 7.4b).[4] This combinator has a `writeD` task at the left-hand side that becomes stable after one rewrite step in which it writes the value to the given pin. When `writeD` becomes stable, the written value is the task value that is observed by the right-hand side of the `>>=.` combinator. Then the interpreter is used again to evaluate the expression, now that the argument of the function is known. The result of the call to the function is again a task tree, but now with different arguments to the tasks, e.g. the state in `writeD` is inversed.

7.3.3 Memory management

The third and final phase is memory management. The mTask RTS is designed to run on systems with as little as *2* KiB of RAM. Aggressive memory management is therefore vital. Not all firmwares for microprocessors support heaps and—when they do—allocation often leaves holes when not used in a *last in first out* strategy. The RTS uses a chunk of memory in the global data segment with its own memory manager tailored to the needs of mTask. The size of this block can be changed in the configuration of the RTS if necessary. On an Arduino UNO—equipped with *2* KiB of RAM—the maximum viable size is about *1500* B. The self-managed memory uses a similar layout as the memory layout for C programs only the heap and the stack are switched (see figure 7.5).

---

[4]`t1 >>|. t2` is a shorthand for `t1 >>*. [IfStable id \_→t2]`.
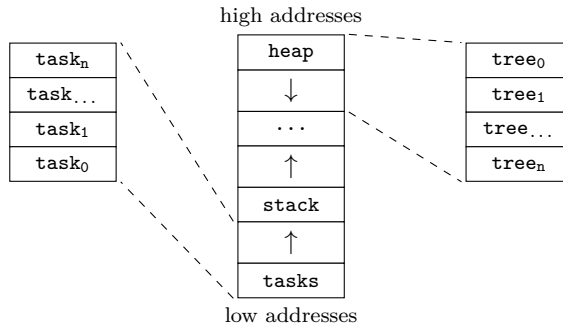
Figure 7.5: Memory layout in the mTask RTS.

A task is stored below the stack and it consists of the task id, a pointer to the task tree in the heap (null if not initialised yet), the current task value, the configuration of SDSs, the configuration of peripherals, the byte code and some scheduling information.

In memory, task data grows from the bottom up and the interpreter stack is located directly on top of it growing in the same direction. As a consequence, the stack moves when a new task is received. This never happens within execution because communication is always processed before execution. Values in the interpreter are always stored on the stack. Compound data types are stored unboxed and flattened. Task trees grow from the top down as in a heap. This approach allows for flexible ratios, i.e. many tasks and small trees or few tasks and big trees.

Stable tasks, and unreachable task tree nodes are removed. If a task is to be removed, tasks with higher memory addresses are moved down. For task trees—stored in the heap—the RTS already marks tasks and task trees as trash during rewriting, so the heap can be compacted in a single pass. This is possible because there is no sharing or cycles in task trees and nodes contain pointers to their parent.

7.4   C code generation for communication

All communication between the iTask server and the mTask server is type parametrised and automated. From the structural representation of the type, a Clean parser and printer is constructed using generic programming. Furthermore, a C/C++ parser and printer is generated for use on the mTask device. The technique for generating the C/C++ parser and printer is very similar to template metaprogramming and requires a rich generic programming library or compiler support that includes a lot of metadata in the record and constructor nodes. Using generic programming in the mTask system, both serialisation and deserialisation on the microcontroller and the server is automatically generated.

7.4.1 Server

On the server, off-the-shelve generic programming techniques are used to make the serialisation and deserialisation functions (see listing 7.14). Serialisation is a simple conversion from a value of the type to a string. Deserialisation is a bit different in order to support streaming.[5] Given a list of available characters, a tuple is always returned. The right-hand side of the tuple contains the remaining characters, the unparsed input. The left-hand side contains either an error or a maybe value. If the value is a `?None`, there was no full value to parse. If the value is a `?Just`, the data field contains a value of the requested type.

```
generic toByteCode   a    :: a → String
generic fromByteCode a *! :: [Char] → (Either String (? a), [Char])
```

Listing (Clean) 7.14: Serialisation and deserialisation functions in Clean.

7.4.2 Client

The RTS of the mTask system runs on resource-constrained microcontrollers and is implemented in portable C/C++. In order to achieve more interoperation safety, the communication between the server and the client is automated, i.e. the serialisation and deserialisation code in the RTS is generated. The technique used for this is very similar to the technique shown in chapter 3. However, instead of using template metaprogramming, a feature Clean lacks, generic programming is used also as a two-stage rocket. In contrast to many other generic programming systems, Clean allows for access to much of the metadata of the compiler. For example, `Cons`, `Object`, `Field`, and `Record` generic constructors are enriched with their arity, names, types, &c. Furthermore, constructors can access the metadata of the objects and fields of their parent records. Using this metadata, generic functions are created that generate C/C++ type definitions, parsers and printers for any first-order Clean type. The exact details of this technique can be found in the future in a paper that is in preparation.

ADTs are converted to tagged unions, newtypes to typedefs, records to structs, and arrays to dynamic size-parametrised allocated arrays. For example, the Clean types in listing 7.15 are translated to the C/C++ types seen in listing 7.16

```
:: T a = A a | B NT {#Char}
:: NT =: NT Real
```

Listing (Clean) 7.15: Simple ADTs in Clean.

```
typedef double Real;
typedef char Char;

typedef Real NT;
enum T_c {A_c, B_c};
```

---

[5]Here the `*!` variant of the generic interface is chosen that has less uniqueness constraints for the compiler-generated adaptors (Alimarine, 2005; Hinze and Peyton Jones, 2001).

```
struct Char_HshArray { uint32_t size; Char *elements; };
struct T {
    enum T_c cons;
    struct { void *A;
             struct { NT f0; struct Char_HshArray f1; } B;
    } data;
};
```

Listing (C++) 7.16: Generated C/C++ type definitions for the simple ADTs.

For each of these generated types, two functions are created, a typed printer, and a typed parser (see listing 7.17). The parser functions are parametrised by a read function, an allocation function and parse functions for all type variables. This allows for the use of these functions in environments where the communication is parametrised and the memory management is self-managed such as in the mTask RTS.

```
struct T parse_T(uint8_t (*get)(), void *(*alloc)(size_t),
                 void *(*parse_0)(uint8_t (*)(), void *(*)(size_t)));

void print_T(void (*put)(uint8_t), struct T r,
             void (*print_0)(void (*)(uint8_t), void *));
```

Listing (C++) 7.17: Printer and parser for the ADTs in C/C++.

7.5   Conclusion

This chapter showed the implementation of the mTask byte code compiler, the RTS, and the internals of their communication. It is not straightforward to execute mTask tasks on resources-constrained IoT edge devices. To achieve this, the terms in the DSL are compiled to compact domain-specific byte code. This byte code is sent for interpretation to the light-weight RTS of the edge device. The RTS first evaluates the main expression in the interpreter. The result of this evaluation, a run time representation of the task, is a task tree. This task tree is rewritten according to small-step reduction rules until a stable value is observed. Rewriting multiple tasks at the same time is achieved by interleaving the rewrite steps, resulting in seemingly parallel execution of the tasks. All communication, including the serialisation and deserialisation, between the server and the RTS is automated. From the structural representation of the types, printers and parsers are generated for the server and the client.

Chapter 8

Green computing with mTask

This chapter demonstrates the energy-saving features of mTask by:

- giving an overview of general green computing measures for edge devices;
- explaining task scheduling in mTask, and how to tweak it so suit the applications and energy needs;
- showing how to use interrupts in mTask to reduce the need for polling.

The edge layer of the IoT is built from energy-efficient devices that sense and interact with the world. While individual devices consume little energy, the sheer number of devices in total amounts to a lot. Furthermore, many of these devices operate on batteries and higher energy consumption increases the amount of e-waste as IoT edge devices are often hard to reach and consequently hard to replace (Nižetić et al., 2020). It is therefore crucial to lower their energy consumption.

To reduce the power consumption of an IoT edge device, the specialised low-power sleep modes of the microprocessors can be leveraged. Different sleep modes achieve different power reductions because of their run time characteristics. These specifics range from disabling or suspending the Wi-Fi radio; stop powering (parts) of the RAM; disabling peripherals; or even turning off the processor completely, requiring an external signal to wake up again. Determining exactly when, and for how long it is safe to sleep is expensive in the general case. In practise, it means that either annotations in the source code, a RTOS, or a scheduler is required.

Table 8.1 shows the properties and current consumption of two commonly used microcontrollers in their various sleep modes. It uncovers that switching the Wi-Fi radio off yields the biggest energy saving. In most IoT applications, we need Wi-Fi

Table 8.1: Current use (mA) of two microprocessor boards in various sleep modes.

	WEMOS D1 mini				Adafruit Feather M0 Wi-Fi			
	active	modem sleep	light sleep	deep sleep	active	modem sleep	light sleep	deep sleep
Wi-Fi	on	off	off	off	on	off	off	off
CPU	on	on	pend.	off	on	on	idle	idle
RAM	on	on	on	off	on	on	on	on
current	100 to 240	15	0.5	0.002	90 to 300	5	2	0.005

for communications. It is fine to switch it off when not communicating, but after switching it on, the Wi-Fi protocol needs to transmit a number of messages to re-establish the connection. This implies that it is only worthwhile to switch the radio off when this can be done for some time. The details vary per system and situation. As a rule of thumb, derived from experimentation, it is only worthwhile to switch the Wi-Fi off when it is not needed for at least some tens of seconds.

8.1   Green IoT computing

The data in table 8.1 shows that it is worthwhile to put the system in some sleep mode when there is temporarily no work to be done. A deeper sleep mode saves more energy, but also requires more work to restore the software to its working state. A processor like the ESP8266 driving the WEMOS D1 mini loses the content of its RAM in deep sleep mode. As a result, after waking up, the program itself is preserved, since it is stored in flash memory, but the program state is lost. When there is a program state to be preserved, we must either store it elsewhere, limit us to light sleep, or use a microcontroller that keeps the RAM intact during deep sleep. For mTasks this implies that the mTask RTS is preserved during deep sleep, but all shipped tasks and their states are lost.

For edge devices executing a single task, explicit sleeping to save energy can be achieved without too much hassle. This becomes much more challenging as soon as multiple independent tasks run on the same device. Sleeping of the device induced by one task prevents progress of all tasks. This is especially annoying when the other tasks are executing time critical parts, like communication protocols. Such protocols control the communication with sensors and actuators. Without the help of an OS, the programmer is forced to combine all subtasks into one big system that decides if it is safe to sleep for all subtasks.

The mTask language offers abstractions for edge layer-specific details such as the heterogeneity of architectures, platforms and frameworks; peripheral access; and multitasking but also for energy consumption and scheduling. In the mTask system, tasks are implemented as a rewrite system, where the work is automatically segmented in small atomic bits and stored as a task tree. Each cycle, a single rewrite step is performed on all task trees. During rewriting, each step, tasks do a bit of their work and progress steadily, allowing interleaved and seemingly parallel

operation. Atomic blocks, such as I/O, are always contained within a rewrite step. This is very convenient, since the system can inspect the current state of all mTask expressions after a rewrite and decide if sleeping and how long is possible. After each loop, the RTS knows which task is waiting on which triggers. With this information, the system determines when it is possible and safe to sleep, and chooses the optimal sleep mode according to the sleeping time.

8.2 Rewrite interval

Some mTask programs contain one or more explicit `delay` primitives, offering a natural pause. However, there are many mTask programs that just specify a repeated set of primitives. A typical example is the program that reads the temperature for a sensor and sets the system LED if the reading is below some given `goal`.

```
thermostat :: Main (MTask v Bool) | mtask, dht v
thermostat = declarePin D8 PMOutput \ledPin→
    DHT (DHT_DHT (i2c 0x36)) \dht→
    {main = rpeat (temperature dht >>~. \temp→
                   writeD ledPin (goal <. temp))}
```

Listing (Clean) 8.1: A basic thermostat task.

This program repeatedly reads the DHT sensor and sets the on-board LED based on the comparison with the `goal` as fast as possible. The mTask machinery ensures that if there are other tasks running on the device, they make progress. However, this solution is far from perfect when we take power consumption into account. In most applications, it is very unlikely that the temperature changes significantly within one minute, let alone within some milliseconds. Hence, it is sufficient to repeat the measurement with an appropriate interval.

There are various ways to improve this program. The simplest solution is to add an explicit delay to the body of the repeat loop. A slightly more sophisticated option is to add a repetition period to the `rpeat` combinator. The combinator implementing this is called `rpeatEvery`. Both solutions rely on an explicit action of the programmer.

Fortunately, mTask also contains machinery to do this automatically. The key of this solution is to associate an evaluation interval with each task dynamically. The interval ⟨*low*, *high*⟩ indicates that the evaluation can be safely delayed by any number of milliseconds within that range. Such an interval is just a hint for the RTS. It is not a guarantee that the evaluation takes place in the given interval. For example, other parts of the task expression can force an earlier evaluation of this part of the task. On the other hand, when the system is very busy with other work, the task might even be executed after the upper bound of the interval. The system calculates the rewrite rates from the current state of the task, i.e. the task tree. This has the advantage that the programmer does not have to deal with them explicitly and that they are available in each and every mTask program.

Table 8.2: Default rewrite rates of basic tasks.

task	default interval
reading an SDS	$\langle 0, 2000 \rangle$
slow sensor	$\langle 0, 2000 \rangle$
medium sensor	$\langle 0, 1000 \rangle$
fast sensor	$\langle 0, 100 \rangle$

$$\mathcal{R} :: (MTask\ v\ a) \rightarrow \langle Int, Int \rangle$$

$$\mathcal{R}(t_1\ .||.\ t_2) = \mathcal{R}(t_1) \cap_{safe} \mathcal{R}(t_2) \tag{8.1}$$

$$\mathcal{R}(t_1\ .\&\&.\ t_2) = \mathcal{R}(t_1) \cap_{safe} \mathcal{R}(t_2) \tag{8.2}$$

$$\mathcal{R}(t \gg *.\ [a_1 \ldots a_n]) = \mathcal{R}(t) \tag{8.3}$$

$$\mathcal{R}(rpeat\ t\ start) = \begin{cases} \mathcal{R}(t) & \text{if } t \text{ is unstable} \\ \langle r_1 - start, r_2 - start \rangle & \text{otherwise} \\ & \textbf{where } \mathcal{R}(t) = \langle r_1, r_2 \rangle \end{cases} \tag{8.4}$$

$$\mathcal{R}(waitUntil\ d) = \langle e - time, e - time \rangle \tag{8.5}$$

$$\mathcal{R}(t) = \begin{cases} \langle \infty, \infty \rangle & \text{if } t \text{ is Stable} \\ \langle r_l, r_u \rangle & \text{otherwise} \end{cases} \tag{8.6}$$

Definition 8.1: Function $\mathcal{R}$ for deriving refresh rates.

8.2.1 Basic tasks

We start by assigning default rewrite rates to basic tasks. These rewrite rates reflect the expected change rates of sensors and other inputs. Basic tasks to set a value of a sensor or actuator have a rate of $\langle 0, 0 \rangle$, i.e. this is never delayed. An example of such a one-shot task in the task that writes to a GPIO pin. Basic tasks that continuously read a sensor or otherwise interact with a peripheral have default rewrite rates that fit standard usage of the sensor. Table 8.2 shows the default values for the basic tasks. Reading SDSs and measuring fast sensors such as sound or light aim for a rewrite every *100* ms. Medium slow sensors such as gesture sensors are expected to rewrite every *1000* ms. Slow sensors such as temperature or air quality have an upper bound of *2000* ms. In section 8.2.4 we show how the programmer can tweak these rewrite rates to match specific needs.

8.2.2 Deriving rewrite rates

Based on these default rewrite rates, the system automatically derives rewrite rates for composed mTask expressions using the function $\mathcal{R}$ as shown in definition 8.1.

$$X \cap_{safe} Y = \begin{cases} X \cap Y & X \cap Y \neq \emptyset \\ Y & Y_2 < X_1 \\ X & \text{otherwise} \end{cases} \quad \textbf{where } Y = \langle Y_1, Y_2 \rangle \text{ and } X = \langle X_1, X_2 \rangle$$

<div align="center">Definition 8.2: Safe intersection operator</div>

Parallel combinators

For parallel combinators, the disjunction combinator (`.||.`) in equation 8.1 and the conjunction combinator (`.&&.`) in equation 8.2, the safe intersection (see definition 8.2) of the rewrite rates is taken to determine the rewrite rate of the complete task. The conventional intersection does not suffice here because it yields an empty intersection when the intervals do not overlap. In that case, the safe intersection returns the range with the lowest numbers. The rationale is that subtasks should preferably not be delayed longer than their rewrite range. Evaluating a task earlier should not change its result but just consumes more energy.

Sequential combinators

For the step combinator (equation 8.3)—and all other derived sequential combinators—the refresh rate of the left-hand side task is taken since that is the only task that is rewritten during evaluation. Only after stepping, the combinator rewrites to the result of evaluating the right-hand side expression.

Repeating combinators

The repeat combinator repeats its argument indefinitely. There are two repeating combinators, `rpeat` and `rpeatEvery` that both use the same task tree node. The `rpeat` task combinator is a special type of `rpeatEvery`, i.e. the rewrite rate is fixed to $\langle 0, 0 \rangle$. The derived refresh rate of the repeat combinator is the refresh rate of the child if it is unstable. Otherwise, the refresh rate is the embedded rate time minus the start time. In case of the `rpeat` task, the default refresh rate is $\langle 0, 0 \rangle$ so the task immediately refreshes and starts the task again.

Delay combinators

Upon installation, a `delay` task is stored as a `waitUntil` task, containing the time of installation added to the specified time to wait. Execution wise, it waits until the current time exceeds the time the argument time.

Other tasks

All other tasks are captured by equation 8.6. If the task is stable, rewriting can be delayed indefinitely since the value will not change anyway. In all other cases, the values from table 8.2 apply where r_l and r_u represent the lower and upper bound of this rate.

Table 8.3: Rewrite steps of the thermostat from listing 8.1 and associated intervals.

Step	Expression	Interval

0	`rpeat ( temperature dht >>~. \temp.`
	`        writeD builtInLED (goal <. temp)`
	`)`
		$\langle 0, 0 \rangle$

1	`temperature dht >>~. \temp.`
	`writeD builtInLED (goal <. temp) >>|.`
	`rpeat ( temperature dht >>~. \temp.`
	`        writeD builtInLED (goal <. temp)`
	`)`
		$\langle 0, 2000 \rangle$

2	`writeD builtInLED false >>|.`
	`rpeat ( temperature dht >>~. \temp.`
	`        writeD builtInLED (goal <. temp)`
	`)`
		$\langle 0, 0 \rangle$

8.2.3   Example

The rewrite intervals associated with various steps of the thermostat program from listing 8.1 are given in table 8.3. The rewrite steps and intervals are circular. After step 2 we continue with step 0 again. Only the actual reading of the sensor with `temperature dht` offers the possibility for a non-zero delay.

8.2.4   Tweaking rewrite rates

A tailor-made ADT (see listing 8.2) is used to tweak the timing intervals. The value is determined at runtime, but the constructor is known at compile time. During compilation, the constructor of the ADT is checked and code is generated accordingly. If it is `Default`, no extra code is generated. In the other cases, code is generated to wrap the task tree node in a *tune rate* node. In the case that there is a lower bound, i.e. the task must not be executed before this lower bound, an extra *rate limit* task tree node is generated that performs a no-op rewrite if the lower bound has not passed but caches the task value.

```
:: TimingInterval v = Default
               | BeforeMs (v Int)          // yields ⟨0, x⟩
               | BeforeS  (v Int)          // yields ⟨0, x × 1000⟩
               | ExactMs  (v Int)          // yields ⟨x, x⟩
               | ExactS   (v Int)          // yields ⟨0, x × 1000⟩
               | RangeMs  (v Int) (v Int)  // yields ⟨x, y⟩
               | RangeS   (v Int) (v Int)  // yields ⟨x × 1000, y × 1000⟩
```

Listing (Clean) 8.2: The ADT for timing intervals in mTask.

Sensors and shared data sources

In some applications, it is necessary to read sensors or SDSs at a different rate than the default rate given in table 8.2, i.e. to customise the rewrite rate. This is achieved by calling the access functions with a custom rewrite rate as an additional argument (suffixed with the backtick (`)) The adaptions to other classes are similar and omitted for brevity. Listing 8.3 shows the extended `dht` and `dio` class definition with functions for custom rewrite rates.

```
class dht v where
    ...
    temperature` :: (TimingInterval v) (v DHT) → MTask v Real
    temperature  ::                     (v DHT) → MTask v Real
    humidity`    :: (TimingInterval v) (v DHT) → MTask v Real
    humidity     ::                     (v DHT) → MTask v Real

class dio p v | pin p where
    ...
    readD` :: (TimingInterval v) (v p) → MTask v Bool | pin p
    readD  ::                     (v p) → MTask v Bool | pin p
```

Listing (Clean) 8.3: Auxiliary definitions to listings 5.14 and 5.16 for DHT sensors and digital GPIO with custom timing intervals.

As an example, we define an mTask that updates the SDS `tempSds` in iTask in a tight loop. The `temperature`` reading dictates that this happens at least once per minute. Without other tasks on the device, the temperature SDS is updated once per minute. Other tasks can cause a slightly more frequent update.

```
delayTime :: TimingInterval v | mtask v
delayTime = BeforeS (lit 60) // 1 minute in seconds

devTask :: Main (MTask v Real) | mtask, dht, lowerSds v
devTask =
    DHT (DHT_DHT pin DHT11) \dht =
    lowerSds \localSds = tempSds
    In {main = rpeat (temperature` delayTime dht >>~. setSds localSds)}
```

Listing (Clean) 8.4: Updating an SDS in iTask at least once per minute.

Repeating tasks

The task combinator `rpeat` restarts the child task in the evaluation if the previous produced a stable result. However, in some cases it is desirable to postpone the restart of the child. For this, the `rpeatEvery` task is introduced which receives an extra argument, the rewrite rate, as shown in listing 8.5. Instead of immediately restarting the child once it yields a stable value, it checks whether the lower bound of the provided timing interval has passed since the start of the task.[1]

---

[1]In reality, it also compensates for time drift by taking into account the upper bound of the timing interval. If the task takes longer to stabilise than the upper bound of the timing interval, this upper bound is taken as the start of the task instead of the actual start.

```
class rpeat v where
    rpeat :: (MTask v t) → MTask v t
    rpeatEvery v :: (TimingInterval v) (MTask v t) → MTask v t
```

Listing (Clean) 8.5: Repeat task combinator with a timing interval.

Listing 8.6 shows an example of an mTask task utilising the `rpeatEvery` combinator that would be impossible to create with the regular `rpeat`. The `timedPulse` function creates a task that sends a *50* ms pulse to the GPIO pin 0 every second. The task created by the `timedPulseNaive` functions emulates the behaviour by using `rpeat` and `delay`. However, this results in a time drift because rewriting tasks trees takes some time and the time it takes can not always be reliably predicted due to external factors. E.g. writing to GPIO pins takes some time, interrupts may slow down the program (see section 8.4), or communication may occur in between task evaluations.

```
timedPulse :: Main (MTask v Bool) | mtask v
timedPulse = declarePin D0 PMOutput \d0→
    in {main = rpeatEvery (ExactSec (lit 1)) (
            writeD d0 true
        >>|. delay (lit 50)
        >>|. writeD d0 false
        )
    }

timedPulseNaive :: Main (MTask v Bool) | mtask v
timedPulseNaive = declarePin D0 PMOutput \d0→
    {main = rpeat (
            writeD d0 true
        >>|. delay (lit 50)
        >>|. writeD d0 false
        >>|. delay (lit 950))
    }
```

Listing (Clean) 8.6: Example program for the repeat task combinator with a timing interval.

8.3   Task scheduling in the mTask engine

The rewrite rates from the previous section only tell us how much the next evaluation of the task can be delayed. In the mTask system, an IoT edge device can run multiple tasks. In addition, it has to communicate with a server to collect new tasks and updates of SDSs. Hence, the rewrite intervals cannot be used directly to let the microcontroller sleep, so a scheduler is involved. Our scheduler has the following objectives.

- Meet the deadline whenever possible, i.e. the system tries to execute every task before the end of its rewrite interval. Only too much work on the device might cause an overflow of the deadline.

- Achieve long sleep times. Waking up from sleep consumes some energy and takes some time. Hence, we prefer a single long sleep over splitting the sleep interval into several smaller pieces.
- The scheduler tries to avoid unnecessary evaluations of tasks as much as possible. A task should not be evaluated now when its execution can also be delayed until the next time that the device is active. That is, a task should preferably not be executed before the start of its rewrite interval. Whenever possible, task execution should even be delayed when we are inside the rewrite interval as long as we can execute the task before the end of the interval.
- The optimal power state should be selected. Although a system uses less power in a deep sleep mode, it also takes more time and energy to wake up from deep sleep. When the system knows that it can sleep only for a short time it is better to go to light sleep mode since waking up from light sleep is faster and consumes less energy.

The algorithm $\mathcal{R}$ from section 8.2.2 computes the evaluation rate of the current tasks. For the scheduler, we transform this interval to an absolute evaluation interval; the lower and upper bound of the start time of that task measured in the time of the IoT edge device. We obtain those bounds by adding the current system time to the bounds of the computed rewrite interval by algorithm $\mathcal{R}$.

For the implementation, it is important to note that the evaluation of a task takes time. Some tasks are extremely fast, but other tasks require longer computations and time-consuming communication with peripherals as well as with the server. These execution times can yield a considerable and noticeable time drift in mTask programs. For instance, a task like `rpeatEvery (ExactMs 1) t` should repeat `t` every millisecond. The programmer might expect that `t` will be executed for the $(N + 1)$th time after N milliseconds. Uncompensated time drift makes this considerably later. The mTask RTS does not pretend to be a hard RTOS, and gives no firm guarantees with respect to evaluation time. Nevertheless, we try to make time handling as reliable as possible. This is achieved by adding the start time of this round of task evaluations rather than the current time to compute absolute execution intervals.

8.3.1   Scheduling Tasks

Apart from the task to execute, the device maintains the connection with the server and checks there for new tasks and updates of SDS. When the microcontroller is active, the connection is checked and updates from the server are processed. After that, the tasks that are within the execution window are executed. Next, the microcontroller goes to light sleep for the minimum of a predefined interval and the task delay.

In general, the microcontroller executes multiple mTask tasks at the same time. The mTask device repeatedly checks for inputs from the server and executes all tasks that cannot be delayed to the next evaluation round one step. The tasks are stored in a priority queue to check efficiently which of them need stepping. The mTask tasks are ordered at their latest start time in this queue; earliest deadline first. We use the earliest deadline to order tasks with equal latest deadline.

It is very complicated to make an optimal scheduling algorithm for tasks to minimise the energy consumption. We use a simple heuristic to evaluate tasks and determine sleep time rather than wasting energy on a fancy evaluation algorithm. Algorithm 8.1 gives this algorithm in pseudo code. First the edge device checks for new tasks and updates of SDSs. This communication adds the new task to the queue, if there were any. The `stepped` set contains all tasks evaluated in this evaluation round. Next, we evaluate tasks from the queue until we encounter a task that has an evaluation interval that has not started. This may result in evaluating tasks earlier than required, but maximises the opportunities to sleep after this evaluation round. Executed tasks are temporarily stored in the `stepped` set instead of inserted directly into the queue to ensure that they are evaluated at most once in an evaluation round to ensure that there is frequent communication with the server. A task that produces a stable value is completed and is not queued again.

Data: queue = [];

```
 1 begin
 2   repeat
 3      time = currentTime();
 4      queue += communicateWithServer();
 5      stepped = [];                    // tasks stepped in this round
 6      while ¬empty(queue) ∧ earliestDeadline(top(queue)) ≤ time do
 7         (task, queue) = pop(queue);
 8         task2 = step(task);   // computes new execution interval
 9         if ¬ isStable(task2) then     // not finished after step
10            stepped += task2;
11         end
12      end
13      queue = merge(queue, stepped);
14      sleep(queue);
15   end
16 end
```

Algorithm 8.1: Pseudo code for the evaluation round of tasks in the queue.

The `sleep` function determines the maximum sleep time based on the top of the queue. The computed sleep time and the characteristics of the microprocessor determine the length and depth of the sleep. For very short sleep times it is not worthwhile to put the processor in sleep mode. In the current mTask RTS, the thresholds are determined by experimentation but can be tuned by the programmer. On systems that lose the content of their RAM it is not possible to go to deep sleep mode.

Table 8.4: Overview of GPIO interrupt types.

type	triggers
change	input changes
falling	input becomes low
rising	input becomes high
low	input is low
high	input is high

8.4 Interrupts

Most microcontrollers have built-in support for processor interrupts. These interrupts are hard-wired signals that interrupt the normal flow of the program or sleep state in order to execute a small piece of code, the interrupt service routine (ISR). While the ISRs look like regular functions, they do come with some limitations. For example, they must be very short, in order not to miss future interrupts; can only do very limited I/O; cannot reliably check the clock; and they operate in their own stack, and thus communication must happen via global variables. After the execution of the ISR, the normal program flow is resumed. Interrupts are heavily used internally in the firmware of microcontrollers to perform timing critical operations such as Wi-Fi, I^{2}C, or Serial Peripheral Interface (SPI) communication; completed ADC conversions; software timers; exception handling; *&c.*

Using interrupts in an mTask task offer two substantial benefits: fewer missed events and better energy usage. Sometimes an external event such as a button press only occurs for a small duration, making it possible to miss it due to it happening right between two polls. Using interrupts is not a fool-proof way of never missing an event. Events could still be missed if they occur during the execution of an ISR or while the microcontroller was in the process of waking up from a triggered interrupt. There are also some sensors, such as the CCS811 air quality sensor, with support for triggering interrupts when a measurement exceeds a critical limit.

There are several different types of interrupts possible that each fire in slightly different circumstances (see table 8.4).

8.4.1 Arduino platform

Listing 8.7 shows an exemplatory program utilising interrupts written using the C/C++ dialect of Arduino. The example shows a debounced light switch for the built-in LED connected to GPIO pin 13. When the user presses the button connected to GPIO pin 11, the state of the LED changes. As buttons sometimes induce noise shortly after pressing, events within *30* ms after pressing are ignored. In between the button presses, the device goes into deep sleep using the `LowPower` library to handle the processor specific sleep interface.

Lines 1 to 3 define the pin and debounce constants. Line 5 defines the current state of the LED, it is declared **volatile** to exempt it from compiler optimisations

because it is accessed in the interrupt handler. Line 6 flags whether the program is in debounce state, i.e. events should be ignored for a short period of time.

In the `setup` function (lines 8 to 12), the pinmode of the LED and interrupt pins are set. Furthermore, the microcontroller is instructed to wake up from sleep mode when a *rising* interrupt occurs on the interrupt pin and to call the ISR at lines 21 to 25. This ISR checks if the program is in cooldown state. If this is not the case, the state of the LED is toggled. In any case, the program goes into cooldown state afterwards.

In the `loop` function, the microcontroller goes to low-power sleep immediately and indefinitely. Only when an interrupt triggers, the program continues, writes the state to the LED, waits for the debounce time, and finally disables the `cooldown` state.

```cpp
1   #define LEDPIN 13
2   #define INTERRUPTPIN 11
3   #define DEBOUNCE 30
4
5   volatile int state = LOW;
6   volatile bool cooldown = true;
7
8   void setup() {
9       pinMode(LEDPIN, OUTPUT);
10      pinMode(INTERRUPTPIN, INPUT);
11      LowPower.attachInterruptWakeup(INTERRUPTPIN, buttonPressed, RISING);
12  }
13
14  void loop() {
15      LowPower.sleep();
16      digitalWrite(LEDPIN, state);
17      delay(DEBOUNCE);
18      cooldown = false;
19  }
20
21  void buttonPressed() { /* ISR */
22      if (!cooldown)
23          state = !state;
24      cooldown = true;
25  }
```

Listing (C++) 8.7: Light switch using interrupts.

8.4.2   The mTask language

Listing 8.8 shows the interrupt interface in mTask. The `interrupt` class contains a single function that, given an interrupt mode and a GPIO pin, produces a task that represents this interrupt. Lowercase variants of the various interrupt modes such as `change :== lit Change` are available as convenience macros (see section 5.3). When the mTask device executes this task, it installs an ISR and sets the rewrite rate of the task to infinity, $\langle \infty, \infty \rangle$. The interrupt handler is set up in such a way that

the rewrite rate is changed to $\langle 0, 0 \rangle$ once the interrupt triggers. As a consequence, the task is executed on the next execution cycle.

```
class interrupt v where
    interrupt :: (v InterruptMode) (v p) → MTask v Bool | pin p

:: InterruptMode = Change | Rising | Falling | Low | High
```

<center>Listing (Clean) 8.8: The interrupt interface in mTask.</center>

The `pirSwitch` function in listing 8.9 creates, given an interval in milliseconds, a task that reacts to motion detection by a passive infrared (PIR) sensor (connected to GPIO pin 0) by lighting the LED connected to GPIO pin 13 for the given interval. The system turns on the LED again when there is still motion detected after this interval. By changing the interrupt mode in this program text from `high` to `rising` the system lights the LED only one interval when it detects motion, no matter how long this signal is present at the PIR pin.

```
pirSwitch :: Int → Main (MTask v Bool) | mtask v
pirSwitch =
    declarePin D13 PMOutput \ledpin→
    declarePin D0 PMInput \pirpin→
    {main = rpeat (     interrupt high pirpin
                   >>|. writeD ledpin false
                   >>|. delay (lit interval)
                   >>|. writeD ledpin true) }
```

<center>Listing (Clean) 8.9: Example of a toggle light switch using interrupts.</center>

8.4.3   The mTask engine

While interrupt tasks have their own node type in the task tree, they differ slightly from other node types because they require a more elaborate setup and teardown. Enabling and disabling interrupts is done in a general way in which tasks register themselves after creation and deregister after deletion. Interrupts should be disabled when there are no tasks waiting for that kind of interrupt because unused interrupts can lead to unwanted wake-ups, and only one kind of interrupt can be attached to a pin at the time.

Event registration

The mTask RTS contains an event administration to register which task is waiting on which event. During the setup of an interrupt task, the event administration in the mTask RTS is checked to determine whether a new ISR for the particular pin needs to be registered. Furthermore, this registration allows for a quick lookup in the ISR to find the tasks listening to the events. Conversely, during the teardown, the ISR is disabled again when the last interrupt task of that kind is deleted. The registration is light-weight and consists only of an event identifier and task identifier. This event registration is stored as a linked list of task tree nodes so that the garbage collector cleans them up when they become unused.

Registering and deregistering interrupts is a device-specific procedure, although most supported devices use the Arduino API for this. Which pins support which interrupt differs greatly from device to device, but this information is known at compile time. At the time of registration, the RTS checks whether the interrupt is valid and throws an mTask exception if it is not. Moreover, an exception is thrown if multiple types of interrupts are registered on the same pin.

Triggering interrupts

Once an interrupt fires, tasks registered to that interrupt are not immediately evaluated because it is usually not safe. For example, the interrupt could fire in the middle of a garbage collection process, resulting in corrupt memory. Furthermore, to ensure the ISR to be very short, just a flag in the event administration is set to be processed later. Interrupt event flags are processed at the beginning of the event loop, before tasks are executed. For each subscribed task, the task tree is searched for nodes listening for the particular interrupt. When found, the node is flagged and the pin status is written. Afterwards, the evaluation interval of the task is set to $\langle 0, 0 \rangle$ and the task is reinserted at the front of the scheduling queue to ensure rapid evaluation of the task. Finally, the event is removed from the registration and the interrupt is disabled. The interrupt can be disabled as all tasks waiting for the interrupt become stable after firing. More occurrences of the interrupts do not change the value of the task as stable tasks keep the same value forever. Therefore, it is no longer necessary to keep the interrupt enabled, and it is relatively cheap to enable it again if needed in the future. Evaluating an interrupt task node in the task tree is trivial because all the work was already done when the interrupt was triggered. The task emits the status of the pin as a stable value if the information in the task shows that it was triggered. Otherwise, no value is emitted.

8.5 Conclusion

This chapter shows how we can automatically associate execution intervals to tasks. Based on these intervals, we can delay the executions of those tasks. When all task executions can be delayed, the microprocessor executing those tasks can go to sleep mode to reduce its energy consumption. This is a rather difficult problem that must be solved dynamically, since we make no assumptions on the number and nature of the tasks that will be allocated to an IoT device. Furthermore, the execution intervals offer an elegant and efficient way to add interrupts to the language. Those interrupts offer a more elegant and energy efficient implementation of watching an input than polling this input.

The actual reduction of the energy is of course highly dependent on the number and nature of the task shipped to the edge device. Our examples show a reduction in energy consumption of two orders of magnitude (see (Crooijmans, Lubbers and Koopman, 2022)). Those reductions are a necessity for edge devices running on battery power. Given the exploding number of IoT edge devices, such savings are also mandatory for other devices to limit the total power consumption of the IoT.

Chapter 9

Finale

This chapter wraps up the monograph by means of:

- a conclusion;
- an outlook on future work;
- an overview of the related work;
- and a history of the mTask system.

9.1   Finale

Traditionally, the IoT has been programmed using layered, or tiered, architectures. Every layer has its own software and hardware characteristics, resulting in semantic friction. It is hard to orchestrate the smooth cooperation of the individual components, especially during maintenance of the entire IoT application. TOP is a declarative programming paradigm designed to describe multi-tiered interactive systems from a single source. Such a tierless system prevents the orchestration problems of the tiered approach. The type system of the host language checks the iTask and mTask components and their interaction. However, it is not straightforward to run TOP systems on resource-constrained devices such as edge devices.

The mTask system bridges this gap by providing a TOP programming language for edge devices. It is a full-fledged TOP language hosted in a tiny FP language. Besides the usual FP constructs, it contains basic tasks, task combinators, support for sensors and actuators, and interrupts. It integrates seamlessly into iTask, a TOP system for interactive web applications. In iTask, abstractions are available for the gritty details of interactive web applications such as program distribution, web applications, data storage, and user management. The mTask system abstracts away all technicalities specific to edge devices such as communication, abstractions for sensors and actuators, interrupts and (multi) task scheduling. When mTask is used together with iTask, all layers of the IoT application are programmed from a

single declarative specification (see e.g. section 6.5).

Any device equipped with the mTask RTS can be used in the system and dynamically receive tasks for execution. This domain-specific OS only is uploaded once, hence saving precious write cycles on the program memory. The mTask devices are connected to the iTask system at run time using a single function that takes care of all the communication and error handling. Once connected to a device, tasks written in the mTask DSL can be lifted to iTask tasks. The tasks are specified and compiled at run time, i.e. Clean can be used as a macro language for constructing mTask tasks, tailor making them for the current work requirements. When lifted, other tasks in the system can interact with the task through the usual means. Furthermore, iTask SDSs can be lowered to mTask tasks as well, allowing for automatic bidirectional data sharing between mTask tasks and the iTask system irrespective of task relations.

9.2 Related work

The novelties of the mTask system can be compared to existing systems in several categories. It is an interpreted (section 9.2.1) TOP (section 9.2.6) DSL (section 9.2.2) that may seem similar at first glance to functional reactive programming (FRP) (section 9.2.5), it is implemented in a functional language (section 9.2.3) and due to the execution semantics, multitasking is automatically supported (section 9.2.4). Section 10.3 contains an elaborate related work section regarding tierless systems.

9.2.1 Interpretation

There are a myriad of interpreted programming languages available for more powerful edge devices. For example, for the popular ESP8266 chip there are ports of MicroPython, Lua, BASIC, JavaScript and Lisp. All of these languages, except the Lisp dialect uLisp (see section 9.2.3), are imperative and do not support multitasking out of the box. They lay pretty hefty constraints on the memory and as a result do not work on smaller microcontrollers. Another interpretation solution for the tiniest devices is Firmata, a protocol for remotely controlling the microcontroller using a server as the interpreter host (Steiner, 2009). Decker (2015) describes a FlowLog (Nelson et al., 2014) extension to incorporate IoT devices into their tierless sytem for software-defined network controllers in a similar way as firmata. Grebe and Gill (2016) wrapped Firmata in a remote monad for integration with Haskell that allowed imperative code to be interpreted on the microprocessors. Later this system was extended to support multithreading as well, stepping away from Firmata as the basis and using their own RTS (Grebe and Gill, 2019). Both differ from our approach because it is required to mark continuation points by hand and there is no automatic safe data communication.

Baccelli et al. (2018) provide a single language IoT system based on the RIOT OS that allows runtime deployment of code snippets called containers. Both client and server are written in JavaScript. However, there is no integration between the client and the server other than that they are programmed from a single source.

Matè is an example of a tierless framework for sensor networks where devices run a virtual machine using TinyOS for dynamic provisioning (Levis and Culler, 2002).

9.2.2   DSLs

Many DSLs provide higher-level programming abstractions for microcontrollers, for example providing strong typing or memory safety. Examples of this are Copilot (Hess, 2020) and Ivory (Elliott et al., 2015). Both are imperative DSLs embedded in a functional language that compiles to C/C++.

9.2.3   Functional programming

Haenisch (2016) showed that there are major benefits to using functional languages on edge devices. They show that using function languages increased the security and maintainability of the applications. Traditional implementations of general purpose functional languages have high memory requirements rendering them unuseable for resource-constrained computers. There have been many efforts to create a general purpose functional language that does fit in small memory environments, albeit with some concessions. For example, there has been a history of creating tiny Scheme implementations for specific microcontrollers. It started with BIT (Dubé, 2000) that only required *64* KiB of memory, followed by PICBIT (Feeley and Dubé, 2003) and PICOBIT (St-Amour and Feeley, 2009) that lowered the memory requirements even more. Suchocki and Kalvala (2015) created Microscheme, a functional language targeting Arduino compatible microcontrollers. The *BIT languages all compile to assembly while Microscheme compiles to C++. Their implementation leans heavily on C++ lambdas, that are available even on Arduino AVR targets. An interpreted Lisp implementation called uLisp also exists that runs on microcontrollers as small as the Arduino UNO (Johnson-Davies, 2020).

9.2.4   Multitasking

Applications for tiny computers are often parallel in nature. Tasks like reading sensors, watching input devices, operating actuators and maintaining communication are often loosely dependent on each other and are preferably executed in parallel. Microcontrollers often do not benefit from an OS due to memory and processing constraints. Therefore, writing multitasking applications in an imperative language is possible, but the tasks have to be interleaved by hand (Feijs, 2013). This results in hard to maintain, error-prone and unscalable spaghetti code.

There are many solutions to overcome this problem in imperative languages. If the host language is a functional language (e.g. the aforementioned scheme variants) multitasking can be achieved without this burden relatively easy using continuation style multiprocessing (Wand, 1980). Writing in this style is complicated and converting an existing program in this continuation passing style results in relatively large programs. Moreover, there is no built-in thread-safe communication possible between the tasks. A TOP or FRP language is superior to manual threading because the programmer is not required to explicitly define continuation points.

Table 9.1: An overview of imperative multithreading solutions for tiny computers with their relevant characteristics. The characteristics are: sequential computing, local variable support, parallel composition, deterministic execution, bounded execution and safe shared memory (adapted from Sant'Anna et al. (2013, p. 12)).

Language		Complexity				Safety	
Name	Year	SeqCmp	LocVar	ParCmp	DetEx	BndEx	SafeMem
Preemptive	many	●	●	○	○	rt	○
nesC	2003	○	○	○	●	async	○
OSM	2005	○	●	●	○	○	○
Protothreads	2006	●	○	○	●	○	○
TinyThreads	2006	●	●	○	●	○	○
Sol	2007	●	●	●	●	○	○
FlowTask	2011	●	●	○	○	○	○
Ocram	2013	●	●	○	●	○	○
Céu	2013	●	●	●	●	●	●
mTask	2022	●	●	●	●	●[1]	●[2]

[1] Only for tasks, not for expressions.
[2] Using SDSs.

Regular preemptive multithreading is too memory intensive for smaller microcontrollers and therefore not suitable. Manual interleaving of imperative code can be automated to certain extents. Solutions often require an RTOS, have a high memory requirement, do not support local variables, no thread-safe shared memory, no composition, or no events as described in table 9.1. This table extends the comparison table with mTask in the relevant categories.

9.2.5 Functional reactive programming

The TOP paradigm is often compared to FRP because they appear similar. FRP was introduced by Elliott and Hudak (1997). The paradigm strives to make modelling systems safer, more efficient, and composable. The core concepts are behaviours and events. A behaviour is a value that varies over time. Events are happenings in the real world and can trigger behaviours. Events and behaviours may be combined using combinators. Tasks in TOP are also event driven and can be combined with combinators. TOP allows for more complex collaboration patterns than FRP (Stutterheim et al., 2018). Consequently, TOP is unable to provide strong guarantees on memory usage, something FRP is capable of. For example, arrowised FRP can give guarantees on upper memory bounds (Nilsson et al., 2002). The way FRP, and for that matter TOP, systems are programmed stays close to the design when the domain matches the paradigm. The IoT domain seems to suit this style of programming very well in just the device layer but also for entire IoT systems.

For example, Potato is an FRP language for building entire IoT systems using

powerful devices such as the Raspberry Pi leveraging the Erlang virtual machine (VM) (Troyer et al., 2018). It requires client devices to be able to run the Erlang VM which makes it unsuitable for low memory environments. The emfrp language compiles a FRP specification for a microcontroller to C code (Sawada and Watanabe, 2016). The I/O part, the bodies of some functions, still need to be implemented. These I/O functions can then be used as signals and combined as in any FRP language. Due to the compilation to C it is possible to run emfrp programs on tiny computers. However, in contrast to in mTask, the tasks are not interpreted and there is no automated communication with a server. Other examples are CFRP (Suzuki et al., 2017), XFRP (Shibanai and Watanabe, 2018), Juniper (Helbling and Guyer, 2016), Hailstorm (Sarkar and Sheeran, 2020), Haski (Valliappan et al., 2020), arduino-copilot (Hess, 2020).

9.2.6   Task-oriented programming

TOP as a paradigm has proven to be effective for implementing distributed, multi-user applications in many domains. Examples are conference management (Plasmeijer and Achten, 2006), coastal protection (Lijnse et al., 2011), incident coordination (Lijnse et al., 2012), crisis management (Jansen et al., 2010) and telemedicine (van der Heijden et al., 2011). In general, TOP results in a higher maintainability, a high separation of concerns, and more effective handling of interruptions of workflow. IoT applications contain a distributed and multi-user component, but the software on the device mostly follows multiple loosely dependent workflows. The only other TOP language for embedded systems is μTasks (Piers, 2016). It is a non-distributed TOP eDSL hosted in Haskell designed for embedded systems such as payment terminals. They show that applications tend to be able to cope well with interruptions and are more maintainable. However, the hardware requirements for running the standard Haskell system are high.

9.3   Future work

There are many ways of extending the research on the mTask system that also concerns TOP for resource-constrained devices in general.

9.3.1   Security

The IoT has reached the news concerningly many times regarding the lack of security (Alhirabi et al., 2021). The fact that the devices are embedded in the fabric, are hard to reach and thus to update, and can only run limited cryptographic algorithms due to their constrained resources makes security difficult. The security of mTask and the used protocols are deliberately overlooked at the moment. The mTask language and RTS are modular. For example, the communication channels are communication method agnostic and operate through a simple duplex channel interface. It should therefore be fairly easy to apply standard security measures to them by replacing communication methods and applying off-the-shelve authentication and encryption to the protocol. De Boer (2020) did preliminary

research on securing the communication channels, which proved to be possible without many changes in the protocol. Nonetheless, this deserves much more attention. The future and related work for the security of mTask and tierless systems is more thoroughly presented in section 10.3.3.

9.3.2   Advanced edge devices techniques

Edge devices produce a lot of data. It is not always effective to send this data to the server for processing. Leaving the produced data and computations on the edge device is called edge computing (Shi et al., 2016). The mTask system exhibits many properties of edge computing because it is possible to run entire workflows on the device. However, it is interesting to see how far this can be extended. The mTask language is a high-level DSL, so it is obvious to introduce abstractions for edge computations. For example, add TOP support for machine learning on the edge device using TinyML (Sanchez-Iborra and Skarmeta, 2020). Van der Veen (2020) did preliminary work for embedding bounded datastructures such as arrays to the language. This could be continued and extended with support for sum types.

Another recent advance in IoT edge device programming is battery-less or even battery-free computing. Instead of equipping the edge device with a battery, a capacitor is used in conjunction with energy harvesting systems such as a solar panel. After a reset, the program state is resumed from a checkpoint that was stored in some non-volatile memory. This technique is called intermittent computing (Hester and Sorber, 2019). Many intermittent computing solutions rely on annotations from the programmer to divide the program into atomic blocks, sometimes called tasks as well. These blocks are marked as such, because in the case of a reset of the system, the work must be done again. Examples of such blocks are I^2C transmissions or calculations that rely on recent sensor data. In mTask, all work expressed by tasks is already split up in atomic pieces of work, i.e. the work is a side effect of rewriting. Furthermore, creating checkpoints is fairly straightforward as mTask tasks do not rely on any global state—all information required to execute a task is stored in the task tree. It is interesting to see what TOP abstractions are useful to support intermittent computing properly and what solutions are required to make it work.

Mesh networks allow for communication not only to and fro the device and server but also between devices. The iTask system already contains primitives for distributed operation. For example, it is possible to run tasks or share data with SDSs on different machines. It is interesting to investigate how this networking technique can be utilised in mTask.

Field-programmable gate arrays (FPGAs) are highly customisable integrated chips consisting of programmable gates. Promising research has gone into translating purely functional code to FPGA configurations (Baaij, 2015). It would be interesting to see how and whether (parts of) TOP programs or the functionality of the mTask OS could be translated to FPGA specifications.

9.3.3   Formal semantics

Semantics allow reasoning about the language and programs in order to do (symbolic) simulation, termination checking, task equivalence, or otherwise. For iTask there have been two attempts to formally specify the language. First Koopman et al. (2011) defined a semantics used for property based testing based on a minimal version of iTask. Then Plasmeijer et al. (2012) formalised iTask by providing an executable semantics for the language. Both semantics are not suitable for formal reasoning due to the complexity. Later, Steenvoorden et al. (2019) created TopHat, a TOP language with a complete formal specification with similar features to mTask (Steenvoorden et al., 2019). Antonova (2022) compared parts of mTask to the semantics of TopHat semantics and created a preliminary semantics for a subset of mTask. Future research into extending the formal semantics of mTask is useful to give more guarantees on mTask programs.

9.3.4   Task-oriented programming

In order to keep the resource constraints low, the mTask language contains only a minimal set of simple task combinators. From these simple combinators, complex collaboration patterns can be expressed. The iTask language is designed exactly the opposite. From just a few super combinators, all other combinators are derived. However, this approach requires a very powerful host language in which task combinators can be defined in terms of the host language. It would be fruitful to investigate which workflows cannot be specified with the limited set of combinators available in mTask. Van der Aalst et al. (2003) defines a benchmark set of workflow patterns. It is interesting to see which patterns can already be implemented with just mTask, which require a round-trip with the server, and what additional combinators would be needed.

Editors are a crucial part of TOP. In mTask, sensors can be seen as read-only shared editors that are updated by the system. It is interesting to investigate how actual interactive editors would fit in mTask. For example, many smartwatches contain touch sensitive screens that could be used to interact with the user in this way. Alternatively, sufficiently powerful edge devices can probably run simple web interfaces as well.

SDSs in iTask have a rich set of combinators to transform and combine the SDSs into new SDS. In mTask, SDSs are typed global variables that may or may not proxy an iTask SDS. It could be interesting to port the SDS combinators to mTask as well, allowing them to be transformed and combined also.

9.3.5   Usability

The promise of DSLs has often been that a domain expert could program with little technical knowledge of the host programming language. Some even propose that a DSL is a UI for domain experts to computation platforms (Barišic et al., 2014). In practise this is not always the case due to crippling syntax and convoluted error messages. Recent approaches in interactive editors for programming language source code such as dynamic editors (Koopman et al., 2021) or typed tree editors such as

Hazelnut (Omar et al., 2017) could prove useful for supporting the DSL programmer in using mTask. If the editor produces correct mTask code by construction, much of the problems could be avoided. In the same respect, as mTask is a tagless-final eDSL and uses HOAS, the error messages are complex and larded with host language features. Much research has gone into simplifying these error messages by translating them to the DSL domain, see for example the work by (Serrano, 2018). De Roos (2020) briefly investigated these methods in their research internship. A future directions could be to extend these findings and apply more eDSL error message techniques on mTask as well.

9.3.6   Language features

The serialisation and deserialisation of data types is automated both on the server and the mTask device using generic programming. Using the structural information of the data type, the code responsible for the functionality is automatically generated. Peripherals are not yet fully integrated in such a way. When a peripheral is added, the programmer has to define the correct byte code, implement the instructions in the interpreter, add task tree nodes, and implement them in the rewrite system. It would be interesting to investigate whether this can be automated or centralised in a way.

More elaborate features in the type systems of modern functional programming languages allow for more type safety. The mTask language relies a lot on these features such as (multi-parameter) type classes and existential data types with class constraints. However, it should be possible to make abstractions over an increasing number of features to make it safer still. For example, the pin mode could be made a type parameter of the GPIO pins, or interrupt handling could be made safer by incorporating the capabilities of the devices in order to reduce run-time errors.

9.3.7   Scheduling

The scheduling in mTask works quite well, but it is not real time. There is a variant of FRP called priority-based FRP (P-FRP) that allows for real-time operation (Belwal et al., 2013). Furthermore, an alternative to reducing the energy consumption by going to sleep is stepping down the processor frequency. So called dynamic voltage and frequency scaling (DVFS) is a scheduling technique that slows down the processor in order to reach the goals as late as possible, reducing the power consumption. Belwal et al. (2013) use P-FRP with DVFS to reduce the energy consumption. It is interesting to investigate the possibilities for DVFS in mTask and TOP in general.

9.4   History of mTask

The development of mTask or its predecessors has been going on for almost seven years now though it really set off during my master's thesis. Many colleagues and students have worked on aspects of the mTask system in collaborations, internships

and Bachelor and Master's theses. This section provides an exhaustive overview of the work on mTask and its predecessors.

9.4.1 Generating C/C++ code

A first throw at a class-based shallow eDSL for microcontrollers was made by Plasmeijer and Koopman (2016). The language was called Arduino DSL (ARDSL) and offered a type safe interface to Arduino C++ dialect. A C++ code generation interpretation was available together with an iTask simulation interpretation. There was no support for tasks nor functions. Some time later in the 2015 CEFP summer school, an extended version was created that allowed the creation of imperative tasks, local SDSs and the usage of functions (Koopman and Plasmeijer, 2019). The name then changed from ARDSL to mTask.

9.4.2 Integration with iTask

Lubbers (2017) extended this in his Master's Thesis by adding integration with iTask and a bytecode compiler to the language. SDS in mTask could be accessed on the iTask server. In this way, entire IoT systems could be programmed from a single source. However, this version used a simplified version of mTask without functions. This was later improved upon by creating a simplified interface where SDSs from iTask could be used in mTask and the other way around (Lubbers et al., 2018). It was shown by Amazonas Cabral de Andrade (2018) that it was possible to build real-life IoT systems with this integration. They did so by creating a functioning prototype of a smart home application. Moreover, a course on the mTask simulator was provided at the 2018 3COWS winter school in Košice, Slovakia (Koopman et al., 2023).

9.4.3 Transition to Task-oriented programming

The mTask language as it is now was introduced in 2018 (Koopman et al., 2018). This paper updated the language to support functions, simple tasks, and SDSs but still compiled to Arduino C++ code. Later the byte code compiler and iTask integration was added to the language (Lubbers et al., 2021). Moreover, it was shown that it is very intuitive to write microcontroller applications in a TOP language (Lubbers et al., 2019). One reason for this is that a lot of design patterns that are difficult using standard means are for free in TOP (e.g. multithreading). In 2019, the 3COWS summer school in Budapest, Hungary hosted a course on developing IoT applications with mTask as well (Lubbers et al., 2023b).

9.4.4 Task-oriented programming

In 2022, the SusTrainable summer school in Rijeka, Croatia hosted a course on developing greener IoT applications using mTask (Lubbers and Koopman, 2022). Several students worked on extending mTask with many useful features: van der Veen (2020) did preliminary work on a green computing analysis, built a simulator, and explored the possibilities for adding bounded datatypes; de Roos explored

beautifying error messages; de Boer (2020) investigated the possibilities for secure communication channels; Crooijmans (2021; 2022) added abstractions for low-power operation to mTask such as hardware interrupts and power efficient scheduling; and Antonova (2022) defined a preliminary formal semantics for a subset of mTask. In 2023, the SusTrainable summer school in Coimbra, Portugal will host a course on mTask.

9.4.5   Using mTask in practice

Funded by the Radboud-Glasgow Collaboration Fund, collaborative work was executed with Phil Trinder, Jeremy Singer, and Adrian Ravi Kishore Ramsingh. An existing smart campus application was developed using mTask and quantitatively and qualitatively compared to the original application that was developed using a traditional IoT stack (Lubbers et al., 2020). This research was later extended to include a four-way comparison: Python, MicroPython, iTask, and mTask (Lubbers et al., 2023c) (see chapter 10). Currently, power efficiency behaviour of traditional versus TOP IoT stacks is being compared as well adding a FreeRTOS, and an Elixir implementation to the mix as well.

Episode III:

Tiered versus Tierless Programming

Chapter 10

Tiered versus tierless programming

IoT software is notoriously complex, usually comprising multiple tiers. Traditionally an IoT developer must use multiple programming languages and ensure that the components interoperate correctly. A novel alternative is to use a single *tierless* language with a compiler that generates the code for each component and their correct interoperation.

We report a systematic comparative evaluation of two tierless language technologies for IoT stacks: one for resource-rich sensor nodes (Clean with iTask), and one for resource-constrained sensor nodes (Clean with iTask and mTask). The evaluation is based on four implementations of a typical smart campus application: two tierless and two Python-based tiered. 1. We show that tierless languages have the potential to significantly reduce the development effort for IoT systems, requiring 70% less code than the tiered implementations. Careful analysis attributes this code reduction to reduced interoperation (e.g. two eDSLs and one paradigm versus seven languages and two paradigms), automatically generated distributed communication, and powerful IoT programming abstractions. 2. We show that tierless languages have the potential to significantly improve the reliability of IoT systems, describing how Clean/iTask/mTask maintains type safety, provides higher order failure management, and simplifies maintainability. 3. We report the first comparison of a tierless IoT codebase for resource-rich sensor nodes with one for resource-constrained sensor nodes. The comparison shows that they have similar code size (within 7%), and functional structure. 4. We present the first comparison of two tierless IoT languages, one for resource-rich sensor nodes, and the other for resource-constrained sensor nodes.

10.1 Introduction

Conventional IoT software stacks are notoriously complex and pose very significant software development, reliability, and maintenance challenges. IoT software architectures typically comprise multiple components organised in four or more tiers or layers (Alphonsa, 2021; Ravulavaru, 2018; Sethi and Sarangi, 2017). This is due to the highly distributed nature of typical IoT applications that must read sensor data from end points (the *perception* layer), aggregate and select the data and communicate over a network (the *network* layer), store the data in a database and analyse it (the *application* layer) and display views of the data, commonly on web pages (the *presentation* layer).

Conventional IoT software architectures require the development of separate programs in various programming languages for each of the components/tiers in the stack. This is modular, but a significant burden for developers, and some key challenges are as follows. 1. Interoperating components in multiple languages and paradigms increases the developer's cognitive load who must simultaneously think in multiple languages and paradigms, i.e. manage significant semantic friction. 2. The developer must correctly interoperate the components, e.g. adhere to the API or communication protocols between components. 3. To ensure correctness the developer must maintain type safety across a range of very different languages and diverse type systems. 4. The developer must deal with the potentially diverse failure modes of each component, and of component interoperation.

A radical alternative development paradigm uses a single *tierless* language that synthesises all components/tiers in the software stack. There are established *tierless* languages for web stacks, e.g. Links (Cooper et al., 2007) or Hop (Serrano et al., 2006). In a tierless language the developer writes the application as a single program. The code for different tiers is simultaneously checked by the compiler, and compiled to the required component languages. For example, Links compiles to HTML and JavaScript for the web client and to SQL on the server to interact with the database system. Tierless languages for IoT stacks are more recent and less common, examples include Potato (Troyer et al., 2018), and Clean with iTask/ mTask (Lubbers et al., 2021).

IoT sensor nodes may be microcontrollers with very limited compute resources, or supersensors: resource-rich single board computers like a Raspberry Pi. A tierless language may target either class of sensor node, and microcontrollers are the more demanding target due to the limited resources, e.g. small memory, executing on bare metal, &c.

Potentially a tierless language both reduces the development effort and improves correctness as correct interoperation and communication is automatically generated by the compiler. A tierless language may, however, introduce other problems. How expressive is the language? That is, can it readily express the required functionality? How maintainable is the software? Is the generated code efficient in terms of time, space, and power?

This chapter reports a systematic comparative evaluation of two tierless language technologies for IoT stacks: one targeting resource-constrained microcontrollers, and the other resource-rich supersensors. The basis of the comparison is four

implementations of a typical smart campus IoT stack (Hentschel et al., 2016). Two implementations are conventional tiered Python-based stacks: Python Raspberry Pi system (PRS) and MicroPython WEMOS system (PWS). The other two implementations are tierless: Clean Raspberry Pi system (CRS) and Clean WEMOS system (CWS). Our work makes the following research contributions, and the key results are summarised, discussed, and quantified in section 10.9.

C1 We show that *tierless languages have the potential to significantly reduce the development effort for IoT systems.* We systematically compare code size (source lines of code (SLOC)) of the four smart campus implementations as a measure of development effort and maintainability (Alpernas et al., 2020; Rosenberg, 1997). The tierless implementations require 70% less code than the tiered implementations. We analyse the codebases to attribute the code reduction to three factors. 1. Tierless languages benefit from reduced interoperation, requiring far fewer languages, paradigms and source code files e.g. CWS uses two languages, one paradigm and three source code files where PWS uses seven languages, two paradigms and 35 source code files (tables 10.2 to 10.4). 2. Tierless languages benefit from automatically synthesised, and hence correct, communication between components that may be distributed. 3. Tierless languages benefit from high-level programming abstractions like compositional and higher-order task combinators (section 10.6).

C2 We show that *tierless languages have the potential to significantly improve the reliability of IoT systems.* We demonstrate how tierless languages preserve type safety, improve maintainability and provide high-level failure management. For example, we illustrate a loss of type safety in PRS. We also critique current tool and community support (section 10.7).

C3 We report *the first comparison of a tierless IoT codebase for resource-rich sensor nodes with one for resource-constrained sensor nodes.* The tierless smart campus implementations have a very similar code size (within 7%), as do the tiered implementations. This suggests that the development and maintenance effort of simple tierless IoT systems for resource-constrained and for resource-rich sensor nodes is similar, as it is for tiered technologies. The percentages of code required to implement each IoT functionality in the tierless Clean implementations is very similar, as it is in the tiered Python implementations. This suggests that the code for resource-constrained and resource-rich sensor nodes is broadly similar in tierless technologies, as in tiered technologies (section 10.6.2)

C4 *We present the first comparison of two tierless IoT languages, one designed for resource-constrained sensor nodes (Clean with iTask and mTask), and the other for resource-rich sensor nodes (Clean with iTask).* We show that the bare metal execution environment enforces some restrictions on mTask although they remain high level. Moreover, the environment conveys some advantages, e.g. better control over timing (section 10.8).

The current work extends (Lubbers et al., 2020) as follows. Contributions C3 and C4 are entirely new, and C1 is enhanced by being based on the analysis of four rather than two languages and implementations.

10.2   Background and related work

10.2.1   University of Glasgow smart campus

The University of Glasgow (UoG) is partway through a ten-year campus upgrade programme, and a key goal is to embed pervasive sensing infrastructure into the new physical fabric to form a *smart campus* environment. As a prototyping exercise, we use modest commodity sensor nodes (i.e. Raspberry Pis) and low-cost, low-precision sensors for indoor environmental monitoring.

Sensor nodes have been deployed into 12 rooms in two buildings. The IoT system has an online data store, providing live access to sensor data through a RESTful API. This allows campus stakeholders to add functionality at a business layer above the layers that we consider here. To date, simple apps have been developed including room temperature monitors and campus utilisation maps (Hentschel et al., 2016). A longitudinal study of sensor accuracy has also been conducted (Harth et al., 2018).

10.2.2   IoT applications

Web applications are necessarily complex distributed systems, with client browsers interacting with a remote web server and data store. Typical IoT applications are even more complex as they combine a web application with a second distributed system of sensor and actuator nodes that collect and aggregate data, operate on it, and communicate with the server.

Both web and IoT applications are commonly structured into tiers, e.g. the classical four-tier Linux, Apache, MySQL and PHP (LAMP) stack. IoT stacks typically have more tiers than web applications, with the number depending on the complexity of the application (Sethi and Sarangi, 2017). While other tiers, like the business layer (Muccini and Moghaddam, 2018) may be added above them, the focus of our study is on programming the lower four tiers of the PRS, CRS, PWS and CWS stacks, as illustrated in figure 10.1.

Perception layer collects the data, interacts with the environment, and consists of devices using light, sound, motion, air quality and temperature sensors.

Network layer replays the communication messages between the sensor nodes and the server through protocols such as MQTT.

Application layer acts as the interface between the presentation layer and the perception layer, storing and processing the data.

Presentation layer utilises web components as the interface between the human and devices where application services are provided.
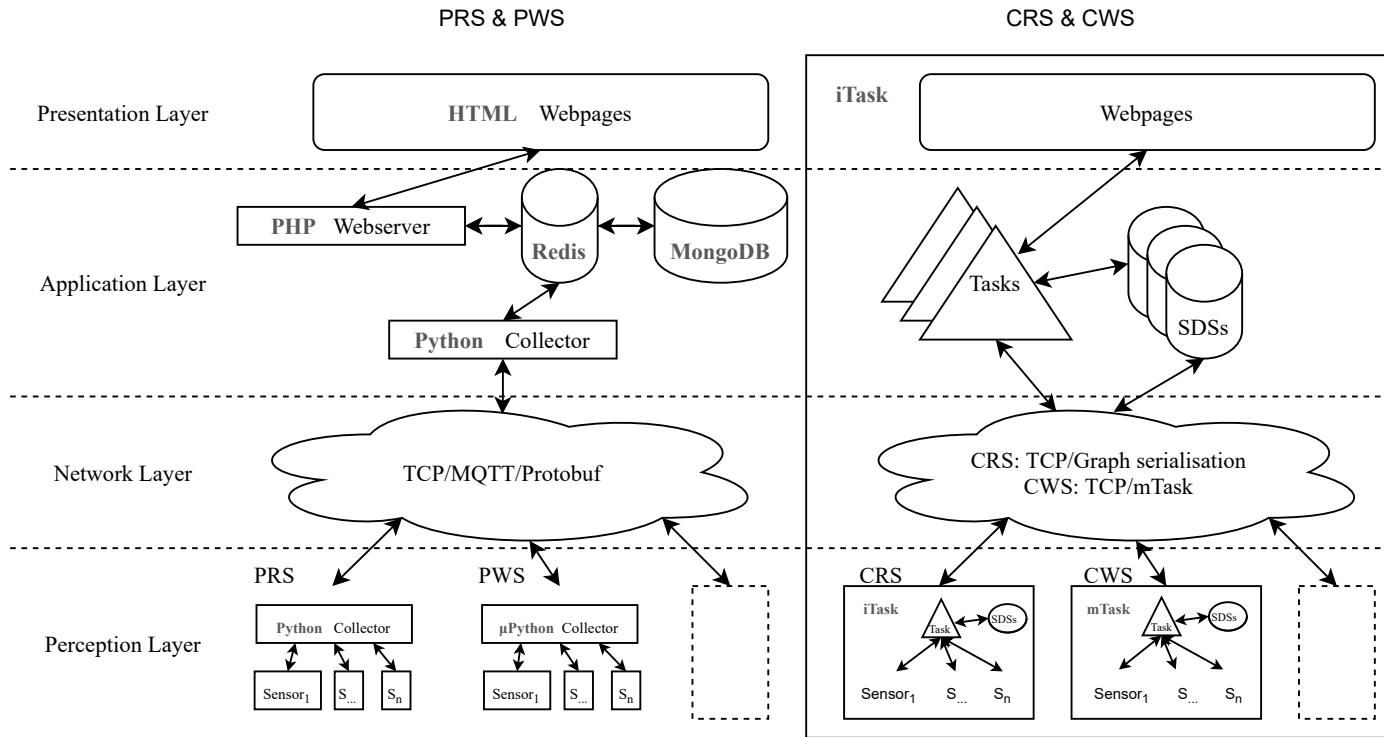
Figure 10.1: PRS and PWS (left) together with CRS and PRS (right) mapped to the four-tier IoT architecture. Every box is the diagram denotes a source file or base. Bold blue text describes the language or technology used in that source. The network and perception layer are unique to the specific implementation, where the application and presentation layers are shared between implementations.

10.2.3 The benefits and challenges of developing tiered IoT stacks

Using multiple tiers to structure complex software is a common software engineering practice that provides significant architectural benefits for IoT and other software. The tiered Python PRS and PWS stacks exhibit these benefits.

Modularity tiers allow a system to be structured as a set of components with clearly defined functionality. They can be implemented independently, and may be interchanged with other components that have similar functionality (MacCormack et al., 2007). In PRS and PWS, for example, a different NoSQL DBMS could relatively easily be substituted for MongoDB.

Abstraction the hierarchical composition of components in the stack abstracts the view of the system as a whole. Enough detail is provided to understand the roles of each layer and how the components relate to one another (Belle et al., 2013). Figure 10.1 illustrates the abstraction of PRS and PWS into four tiers.

Cohesion well-defined boundaries ensure each tier contains functionality directly related to the task of the component (Lee et al., 2001). The tiers in PRS and PWS contain all the functionality associated with perception, networking, application and presentation respectively.

However, a tiered architecture poses significant challenges for developers of IoT and other software. The tiered Python PRS and PWS stacks exhibit these challenges, and we analyse these in detail later in the chapter.

Polyglot development the developer must be fluent in all the languages and components in the stack, known as being a full-stack developer for web applications (Mazzei et al., 2018). That is, the developer must correctly use multiple languages that have different paradigms, i.e. manage significant *semantic friction* (Ireland et al., 2009). For example the PWS developer must integrate components written in seven languages with two paradigms (section 10.6.3).

Correct interoperation the developer must adhere to the API or communication protocols between components. Sections 10.6.1 and 10.6.2 show that communication requires some 17% of PRS and PWS code, so around 100 SLOC. Section 10.6.4 discusses the complexity of writing this distributed communication code.

Maintaining type safety is a key element of the semantic friction encountered in multi-language stacks, and crucial for correctness. The developer must maintain type safety across a range of very different languages and diverse type systems, with minimal tool support. We show an example where PRS loses type safety over the network layer (Section 10.7.1).

Managing multiple failure modes different components may have different failure modes, and these must be coordinated. Section 10.7.2 outlines how PRS and PWS use heartbeats to manage failures.

Beyond PRS and PWS the challenges of tiered polyglot software development are evidenced in real world studies. As recent examples, a study of GitHub open source projects found an average of five different languages in each project, with

many using tiered architectures (Mayer et al., 2017). An earlier empirical study of GitHub shows that using more languages to implement a project has a significant effect on project quality, since it increases defects (Kochhar et al., 2016). A study of IoT stack developers found that interoperation poses a real challenge, that microservices blur the abstraction between tiers, and that both testing and scaling IoT applications to more devices are hard (Motta et al., 2018).

One way of minimising the challenges of developing tiered polyglot IoT software is to standardise and reuse components. This approach has been hugely successful for web stacks, e.g. browser standards. The W3C Web of Things aims to facilitate re-use by providing standardised metadata and other re-useable technological IoT building blocks (Guinard and Trifa, 2016). However, the Web of Things has yet to gain widespread adoption. Moreover, as it is based on web technology, it requires the *thing* to run a web server, significantly increasing the hardware requirements.

10.3   Tierless languages

A radical approach to overcoming the challenges raised by tiered distributed software is to use a tierless programming language that eliminates the semantic friction between tiers by generating code for all tiers, and all communication between tiers, from a single program. Typically a tierless program uses a single language, paradigm and type system, and the entire distributed system is simultaneously checked by the compiler.

There is intense interest in developing tierless, or multi-tiered, language technologies with a number of research languages developed over the last fifteen years, e.g. (Cooper et al., 2007; Ekblad and Claessen, 2014; Serrano et al., 2006; Troyer et al., 2018). These languages demonstrate the advantages of the paradigm, including less development effort, better maintainability, and sound semantics of distributed execution. At the same time a number of industrial technologies incorporate tierless concepts, e.g. (Balat, 2006; Bjornson et al., 2010; Strack, 2015). These languages demonstrate the benefits of the paradigm in practice. Some tierless languages use (embedded) DSLs to specify parts of the multi-tier software.

Tierless languages have been developed for a range of distributed paradigms, including web applications, client-server applications, mobile applications, and generic distributed systems. A recent and substantial survey of these tierless technologies is available (Weisenburger et al., 2020). Here we provide a brief introduction to tierless languages with a focus on IoT software.

10.3.1   Tierless web languages

There are established tierless languages for web development, both standalone languages and DSLs embedded in a host language. Example standalone tierless web languages are Links (Cooper et al., 2007) and Hop (Serrano et al., 2006). From a single declarative program the client, server and database code is simultaneously checked by the compiler, and compiled to the required component languages. For example, Links compiles to HTML and JavaScript for the client side and to SQL

on the server-side to interact with the database system.

An example tierless web framework that uses a DSL is Haste (Ekblad and Claessen, 2014), that embeds the DSL in Haskell. Haste programs are compiled multiple times: the server code is generated by the standard GHC Haskell compiler (Hall et al., 1993); JavaScript for the client is generated by a custom GHC compiler backend. The design leverages Haskell's high-level programming abstractions and strong typing, and benefits from GHC: a mature and sophisticated compiler.

10.3.2 Tierless IoT languages

The use of tierless languages in IoT applications is both more recent and less common than for web applications. Tierless IoT programming may extend tierless web programming by adding network and perception layers. The presentation layer of a tierless IoT language, like tierless web languages, benefits from almost invariably executing in a standard browser. The perception layer faces greater challenges, often executing on one of a set of slow and resource-constrained microcontrollers. Hence, tierless IoT languages typically compile the perception layer to either C/C++ (the lingua franca of microcontrollers), or to some intermediate representation to be interpreted.

DSLs for microcontrollers

Many DSLs provide high-level programming for microcontrollers, for example providing strong typing and memory safety. For example Copilot (Hess, 2020) and Ivory (Elliott et al., 2015) are imperative DSLs embedded in a functional language that compile to C/C++. In contrast to Clean/iTask/mTask such DSLs are not tierless IoT languages as they have no automatic integration with the server, i.e. with the application and presentation layers.

Functional reactive programming

FRP is a declarative paradigm often used for implementing the perception layer of an IoT stack. Examples include mfrp (Sawada and Watanabe, 2016), CFRP (Suzuki et al., 2017), XFRP (Shibanai and Watanabe, 2018), Juniper (Helbling and Guyer, 2016), Hailstorm (Sarkar and Sheeran, 2020), and Haski (Valliappan et al., 2020). None of these languages are tierless IoT languages as they have no automatic integration with the server.

Potato goes beyond other FRP languages to provide a tierless FRP IoT language for resource rich sensor nodes (Troyer et al., 2018). It does so using the Erlang programming language and sophisticated virtual machine.

TOP allows for more complex collaboration patterns than FRP (Stutterheim et al., 2018), and in consequence is unable to provide the strong guarantees on memory usage available in a restricted variant of FRP such as arrowised FRP (Nilsson et al., 2002).

Erlang/Elixir IoT systems

A number of production IoT systems are engineered in Erlang or Elixir, and many are mostly tierless. That is the perception, network and application layers are sets of distributed Erlang processes, although the presentation layer typically uses some conventional web technology. A resource-rich sensor node may support many Erlang processes on an Erlang VM, or low level code (typically C/C++) on a resource-constrained microcontroller can emulate an Erlang process. Only a small fraction of these systems are described in the academic literature, example exceptions are (Shibanai and Watanabe, 2018; Sivieri et al., 2012), with many described only in grey literature or not at all.

10.3.3 Characteristics of tierless IoT languages

This study compares a pair of tierless IoT languages with conventional tiered Python IoT software. Clean/iTask and Clean/iTask/mTask represent a specific set of tierless language design decisions, however many alternative designs are available. Crucially the limitations of the tierless Clean languages, e.g. that they currently provide limited security, should not be seen as limitations of tierless technologies in general. This section briefly outlines key design decisions for tierless IoT languages, discusses alternative designs, and describes the Clean designs. The Clean designs are illustrated in the examples in the following section.

Tier splitting and placement

A key challenge for a tierless language is to determine which parts of the program correspond to a particular tier and hence should be executed by a specific component on a specific host.

For example a tierless web language must identify client code to ship to browsers, database code to execute in the DBMS, and application code to run on the server. Tierless web languages can make this determination statically, so-called *tier splitting* using types or syntactic markers like `server` or `client` pragmas (Cooper et al., 2007; Ekblad and Claessen, 2014). It is even possible to infer the splitting, relieving the developers from the need to specify it, as illustrated for JavaScript as a tierless web language (Philips et al., 2014).

In Clean/iTask/mTask and Clean/iTask tier splitting is specified by functions, e.g. the Clean/iTask/mTask `asyncTask` function identifies a task for execution on a remote device and `liftmTask` executes the given task on an IoT device. The tier splitting functions are illustrated in examples in the next section, e.g. on line 17 in listing 10.3 and line 29 in listing 10.4. Specifying splitting as functions means that new splitting functions can be composed, and that splitting is under program control, e.g. during execution a program can decide to run a task locally or remotely.

As IoT stacks are more complex than web stacks, the *placement* of data and computations onto the devices/hosts in the system is more challenging. In many IoT systems placement is manual: the sensor nodes are microcontrollers that are programmed by writing the program to flash memory. So physical access to

the microcontroller is normally required to change the program, making updates challenging.

Techniques like over-the-air programming and interpreters allow microcontrollers to be dynamically provisioned, increasing their maintainability and resilience. For example Baccelli et al. (2018) provide a single language IoT system based on the RIOT OS that allows runtime deployment of code snippets called containers. Both client and server are written in JavaScript. However, there is no integration between the client and the server other than that they are programmed from a single source. Matè is an example of an early tierless sensor network framework where devices are provided with a virtual machine using TinyOS for dynamic provisioning (Levis and Culler, 2002).

In general different tierless languages specify placement in different ways, e.g. code annotations or configuration files, and at different granularities, e.g. per function or per class (Weisenburger et al., 2020).

Clean/iTask/mTask and Clean/iTask both use dynamic task placement. In Clean/iTask/mTask sensor nodes are programmed once with the mTask RTS, and possibly some precompiled tasks. Thereafter, a sensor node can dynamically receive mTask programs, compiled at runtime by the server. In Clean/iTask the sensor node runs an iTask server that receives and executes code from the (IoT) server (Oortgiese et al., 2017). Placement happens automatically as part of the first-class splitting constructs, so line 29 in listing 10.4 places `devTask` onto the `dev` sensor node.

Communication

Tierless languages may adopt a range of communication paradigms for communicating between components. Different tierless languages specify communication in different ways (Weisenburger et al., 2020). Remote procedures are the most common communication mechanism: a procedure/function executing on a remote host/machine is called as if it was local. The communication of the arguments to, and the results from, the remote procedure is automatically provided by the language implementation. Other mechanisms include explicit message passing between components; publish/subscribe where components subscribe to topics of interest from other components; reactive programming defines event streams between remote components; finally shared state makes changes in a shared and potentially remote data structure visible to components.

Clean/iTask/mTask and Clean/iTask communicate using a combination of remote task invocation, similar to remote procedures, and shared state through SDSs. Listing 10.3 illustrates: line 17 shows a server task launching a remote task, `devTask`, on to a sensor node; and line 19 shows the sharing of the remote `latestTemp` SDS.

Security

Security is a major issue and a considerable challenge for many IoT systems (Alhirabi et al., 2021). There are potentially security issues at each layer in an IoT application (figure 10.1). The security issues and defence mechanisms at the

application and presentation layers are relatively standard, e.g. defending against SQL injection attacks. The security issues at the network and perception layers are more challenging. Resource-rich sensor nodes can adopt some standard security measures like encrypting messages, and regularly applying software patches to the operating system. However, microcontrollers often lack the computational resources for encryption, and it is hard to patch their system software because the program is often stored in flash memory. In consequence there are infamous examples of IoT systems being hijacked to create botnets (Antonakakis et al., 2017; Herwig et al., 2019).

Securing the entire stack in a conventional tiered IoT application is particularly challenging as the stack is implemented in a collection of programming languages with low level programming and communication abstractions. In such polyglot distributed systems it is hard to determine, and hence secure, the flow of data between components. As a consequence, a small mistake may have severe security implications.

A number of characteristics of tierless languages help to improve security. Communication and placement vulnerabilities are minimised as communication and placement are automatically generated and checked by the compiler. So injection attacks and the exploitation of communication/placement protocol bugs are less likely. Vulnerabilities introduced by mismatched types are avoided as the entire system is type checked. Moreover, tierless languages can exploit language level security techniques. For example languages like Jif/split (Zdancewic et al., 2002) and Swift (Chong et al., 2007) place components to protect the security of data. Another example are programming language technologies for controlling information flow, and these can be used to improve security. For example Haski uses them to improve the security of IoT systems (Valliappan et al., 2020).

However, many tierless languages have yet to provide a comprehensive set of security technologies, despite its importance in domains like web and IoT applications. For example Erlang and many Erlang-based systems (Shibanai and Watanabe, 2018; Sivieri et al., 2012), lack important security measures. Indeed, security is not covered in a recent, otherwise comprehensive, survey of tierless technologies (Weisenburger et al., 2020).

Clean/iTask and Clean/iTask/mTask are typical in this respect: little effort has yet been expended on improving their security. Of course as tierless languages they benefit from static type safety and automatically generated communication and placement. Some preliminary work shows that, as the communication between layers is protocol agnostic, more secure alternatives can be used. One example is to run the iTask server behind a reverse proxy implementing TLS/SSL encryption (Wijkhuizen, 2018). A second is to add integrity checks or even encryption to the communication protocol for resource-rich sensor nodes (de Boer, 2020).

10.4   Task-oriented and IoT programming in Clean

To make this chapter self-contained we provide a concise overview of Clean, TOP, and IoT programming in iTask and mTask. The minor innovations reported here

are the interface to the IoT sensors, and the Clean port for the Raspberry Pi.

Clean is a statically typed FP language similar to Haskell: both languages are pure and non-strict (Achten, 2007). A key difference is how state is handled: Haskell typically embeds stateful actions in the `IO` Monad (HaskellWiki contributors, 2020; Peyton Jones and Wadler, 1993). In contrast, Clean has a uniqueness type system to ensure the single-threaded use of stateful objects like files and windows (Barendsen and Smetsers, 1996). Both Clean and Haskell support fairly similar models of generic programming (Rodriguez et al., 2008), enabling functions to work on many types. As we shall see generic programming is heavily used in task-oriented programming (Alimarine and Plasmeijer, 2002; Hinze, 2000), for example to construct web editors and communication protocols that work for any user-defined datatype.

10.4.1   Task-oriented programming

TOP is a declarative programming paradigm for constructing interactive distributed systems (Plasmeijer et al., 2012). Tasks are the basic blocks of TOP and represent work that needs to be done in the broadest sense. Examples of typical tasks range from allowing a user to complete a form, controlling peripherals, moderating other tasks, or monitoring a database. From a single declarative description of tasks all the required software components are generated. This may include web servers, client code for browsers or IoT devices, and for their interoperation. That is, from a single TOP program the language implementation automatically generates an *integrated distributed system.* Application areas range from simple web forms or blinking LEDs to multi-user distributed collaboration between people and machines (Oortgiese et al., 2017).

TOP adds three concepts: tasks, task combinators, and SDSs. Example basic tasks are web editors for user-defined datatypes, reading some IoT sensor, or controlling peripherals like a servo motor. Task combinators compose tasks into more advanced tasks, either in parallel or sequential and allow task values to be observed by other tasks. As tasks can be returned as the result of a function, recursion can be freely used, e.g. to express the repetition of tasks. There are also standard combinators for common patterns. Tasks can exchange information via SDSs (Domoszlai et al., 2014). All tasks involved can atomically observe and change the value of a typed SDS, allowing more flexible communication than with task combinators. SDSs offer a general abstraction of data shared by different tasks, analogous to variables, persistent values, files, databases and peripherals like sensors. Combinators compose SDSs into a larger SDS, and parametric lenses define a specific view on an SDS.

10.4.2   The iTask eDSL

The iTask eDSL is designed for constructing multi-user distributed applications, including web (Plasmeijer et al., 2007a) or IoT applications. Here we present iTask by example, and the first is a complete program to repeatedly read the room temperature from a DHT sensor attached to the machine and display it on a web

page (listing 10.1). The first line is the module name, the third imports the `iTask` module, and the main function (lines 5 and 6) launches `readTempTask` and the iTask system to generate the web interface in figure 10.2.

Interaction with a device like the DHT sensor using a protocol like 1-Wire or I^2C is abstracted into a library. So the `readTempTask` task starts by creating a `dht` sensor object (line 10) thereafter `repeatEvery` executes a task at the specified `interval`. This task reads the temperature from the `dht` sensor, and with a sequential composition combinator `>>~` passes the `temp` value to `viewInformation` that displays it on the web page (line 13). The tuning combinator `<<@` adds a label to the web editor displaying the temperature. Crucially, the iTask implementation transparently ships parts of the code for the web-interface to be executed in the browser, and figure 10.2 shows the UML deployment diagram.

```
1   module simpleTempSensor
2
3   import iTasks
4
5   Start :: *World → *World
6   Start world = doTasks readTempTask world
7
8   readTempTask :: Task Real
9   readTempTask =
10      withDHT IIO_TempID \dht →
11      repeatEvery interval (
12          temperature dht >>~ \temp →
13          viewInformation [] temp <<@
14          Label "Temperature"
15      )
```

Listing (Clean) 10.1: SimpleTempSensor: a Clean/iTask program to read a local room temperature sensor and display it on a web page.

(a) Web page.   (b) Deployment diagram.
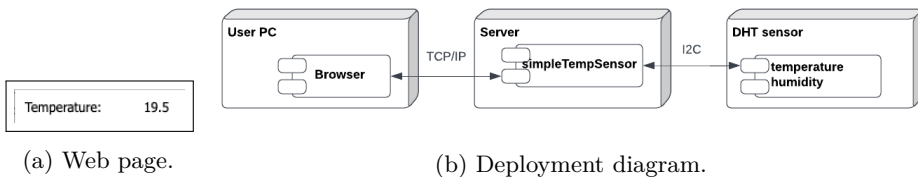
Figure 10.2: SimpleTempSensor written in iTask.

SimpleTempSensor only reports instantaneous temperature measurements. Extending it to store and manipulate timed temperature records produces a tiny tierless web application: TempHistory in listing 10.2. A tierless IoT system can be controlled from a web interface in exactly the same way, e.g. to view and set the measurement frequencies of each of the room sensors. Line 5 defines a record to store `time` and `temp` measurements. Task manipulations are derived for `Measurement` (line 6) and these include displaying measurements in a web-editor and storing

them in a file. Line 8 defines a persistent SDS to store a list of measurements, and for communication between tasks. The SDS is analogous to the SQL DBMS in many tiered web applications.

TempHistory defines two tasks that interact with the `measurementsSDS`: `measureTask` adds measurements at each detected change in the temperature. It starts by defining a `dht` object as before, and then defines a recursive `task` function parameterised by the `old` temperature. This function reads the temperature from the DHT sensor and uses the step combinator, `>>*`, to compose it with a list of actions. The first of those actions that is applicable determines the continuation of this task. If no action is applicable, the task on the left-hand side is evaluated again. The first action checks whether the new temperature is different from the `old` temperature (line 16). If so, it records the current time and adds the new measurements to the `measurementsSDS`. The next action in line 20 is always applicable and waits (sleeps) for an interval before returning the old temperature. On line 22 `task` is launched with an initial temperature.

```
1   module TempHistory
2
3   import iTasks, iTasks.Extensions.DateTime
4
5   :: Measurement = {time :: Time, temp :: Real}
6   derive class iTask Measurement
7
8   measurementsSDS :: SimpleSDSLens [Measurement]
9   measurementsSDS = sharedStore "measurements" []
10
11  measureTask :: Task ()
12  measureTask =
13      withDHT IIO_TempID \dht →
14      let task old =
15              temperature dht >>*
16              [ OnValue (ifValue ((<>) old) \temp →
17                  get currentTime >>~ \time →
18                  upd (\list → [{time = time, temp = temp}:list])
        measurementsSDS
19                  @! temp)
20              , OnValue (always (waitForTimer False interval @! old))
21              ] >>~ task
22      in task initialTemp
23
24
25  controlSDS :: Bool → Task [Measurement]
26  controlSDS byTemp =
27      ((Label "# to take"    @>> enterInformation []) -||
28       (Label "Measurements" @>>
29        viewSharedInformation
30          [ ViewAs (if byTemp (sortBy (\x y → x.temp < y.temp)) id)]
31              measurementsSDS)) >>*
32          [ OnAction (Action "Clear") (always
```

```
33              (set [] measurementsSDS >-|  controlSDS byTemp))
34          , OnAction (Action "Take")  (ifValue ((<) 0)
35              (\n → upd (take n) measurementsSDS >-| controlSDS byTemp))
36          , OnAction (Action (if byTemp "Sort time" "Sort temp")) (always
37              (controlSDS (not byTemp)))
38          ]
39
40 mainTask :: Task [Measurement]
41 mainTask = controlSDS False -|| measureTask
```

Listing (Clean) 10.2: TempHistory: a tierless Clean/iTask web application that records and manipulates timed temperatures.

The `controlSDS` task illustrates communication from the web page user and persistent data manipulation. That is, it generates a web page that allows users to control their view of the temperature measurements, as illustrated in figure 10.3. The page contains 1. a web editor to enter the number `n` of elements to display, generated on line 27. 2. A display of the `n` most recent temperature and time measurements, lines 28 to 31. 3. Three buttons that are again combined with the step combinator `>>*`, lines 31 to 38. The `Clear` button is `always` enabled and sets the SDS to an empty list before calling `controlSDS` recursively. The `Take` button is only enabled when the web editor produces a positive `n` and updates the `measurementsSDS` with the `n` most recent measurements before calling `controlSDS` recursively. The final action is `always` enabled and calls `controlSDS` recursively with the negation of the `byTemp` argument to change the sorting criteria.
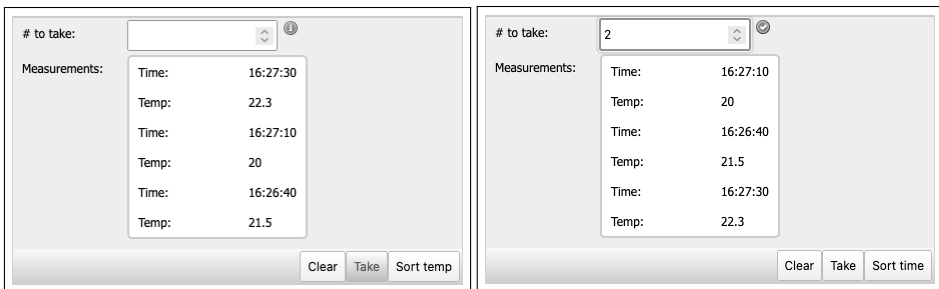
(a) Web page sorted by time.   (b) Web page sorted by temperature.

Figure 10.3: Web pages generated by the `TempHistory` Clean/iTask tierless web application. The `Take` button is only enabled when the topmost editor contains a positive number.

Figure 10.3 shows two screenshots of web pages generated by the `TempHistory` program. Figure 10.4 is the deployment diagram showing the addition of the persistent `measurementsSDS` that stores the history of temperature measurements.
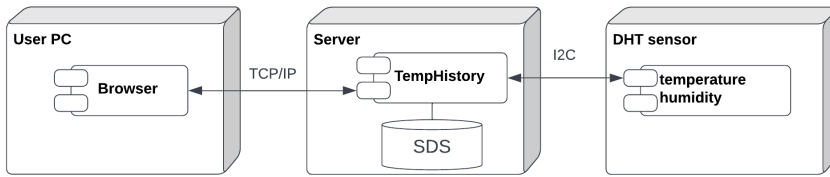
Figure 10.4: Deployment diagram of the iTask TempHistory tierless web application from listing 10.2.

10.4.3   Engineering tierless IoT systems with iTask

A typical IoT system goes beyond a web application by incorporating a distributed set of sensor nodes each with a collection of sensors or actuators. That is, they add the perception and network layers in figure 10.1. If the sensor nodes have the computational resources to support an iTask server, as a Raspberry Pi does, then iTask can also be used to implement these layers, and integrate them with the application and presentation layers tierlessly.

As an example of tierless IoT programming in Clean/iTask listing 10.3 shows a complete temperature sensing system with a server and a single sensor node (Clean Raspberry Pi temperature sensor (CRTS)), omitting only the module name and imports. It is similar to the SimpleTempSensor and TempHistory programs above, for example `devTask` repeatedly sleeps and records temperatures and times, and `mainTask` displays the temperatures on the web page in figure 10.5. There are some important differences, however. The `devTask` (lines 8 to 13) executes on the sensor node and records the temperatures in a standard timestamped (lens on) an SDS: `dateTimeStampedShare latestTemp`. The `mainTask` (line 16) executes on the server: it starts `devTask` as an asynchronous task on the specified sensor node (line 17) and then generates a web page to display the latest temperature and time (lines 18 and 20).

The `tempSDS` is very similar to the `measurementsSDS` from the previous listings. The only difference is that we store measurements as tuples instead of tailor-made records. The `latestTemp` is a *lens* on the `tempSDS`. A lens is a new SDS that is automatically mapped to another SDS. Updating one of the SDSs that are coupled in this way automatically updates the other. The function `mapReadWrite` is parameterised by the read and write functions, the option to handle asynchronous update conflicts (here `?None`) and the SDS to be transformed (here `tempSDS`). The result of reading is the head of the list, if it exists. The type for writing `latestTemp` is a tuple with a new `DateTime` and temperature as `Real`.

```
1  tempSDS :: SimpleSDSLens [(DateTime, Real)]
2  tempSDS = sharedStore "temperatures" []
3
4  latestTemp :: SDSLens () (? (DateTime, Real)) (DateTime, Real)
5  latestTemp = mapReadWrite (listToMaybe, \x xs → ?Just [x:xs]) ?None
        tempSDS
```

```
6
7   devTask :: Task DateTime
8   devTask =
9       withDHT IIO_TempID \dht →
10      forever (
11          temperature dht >>~ \temp →
12          set temp (dateTimeStampedShare latestTemp) >-|
13          waitForTimer False interval)
14
15  mainTask :: Task ()
16  mainTask
17      =   asyncTask deviceInfo.domain deviceInfo.port devTask
18      -|| viewSharedInformation []
19          (remoteShare latestTemp deviceInfo)
20          <<@ Title "Latest temperature"
```

Listing (Clean) 10.3: CRTS: a tierless temperature sensing IoT system. Written in Clean/iTask, it targets a resource-rich sensor node.

(a) Web page.

(b) Deployment diagram.
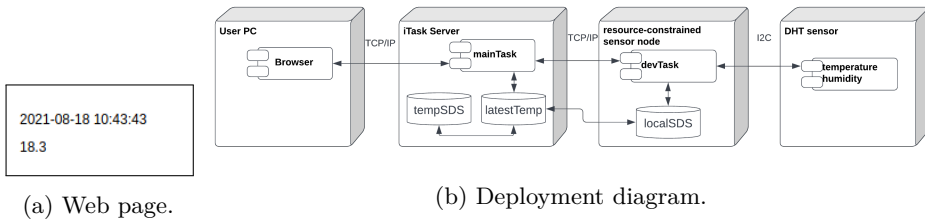
Figure 10.5: Tierless iTask CRTS temperature sensing IoT system.

10.4.4 The mTask eDSL

In many IoT systems the sensor nodes are resource constrained, e.g. inexpensive microcontrollers. These are far cheaper, and consume far less power, than a single-board computer like a Raspberry Pi. Microcontrollers also allow the programmer to easily control peripherals like sensors and actuators via the I/O pins of the processor.

Microcontrollers have limited memory capacity, compute power and communication bandwidth, and hence typically no OS. These limitations make it impossible to run an iTask server: there is no OS to start the remote task, the code of the task is too big to fit in the available memory and the microcontroller processor is too slow to run it. The mTask eDSL is designed to bridge this gap: mTask tasks can be communicated from the server to the sensor node, to execute within the limitations of a typical microcontroller, while providing programming abstractions that are consistent with iTask.

Like iTask, mTask is task oriented, e.g. there are primitive tasks that produce intermediate values, a restricted set of task combinators to compose the tasks, and

(recursive) functions to construct tasks. Tasks in mTask communicate using task values or SDSs that may be local or remote, and may be shared by some iTask tasks.

Apart from the eDSL, the mTask system contains a feather-light domain-specific *operating system* running on the microcontroller. This OS task scheduler receives the byte code generated from one or more mTask programs and interleaves the execution of those tasks. The OS also manages SDS updates and the passing of task results. The mTask OS is stored in flash memory while the tasks are stored in RAM to minimise wear on the flash memory. While sending byte code to a sensor node at runtime greatly increases the amount of communication, this can be mitigated as any tasks known at compile time can be preloaded on the microcontroller. In contrast, compiled programs, like C/C++, are stored in flash memory and there can only ever be a few thousand programs uploaded during the lifetime of the microcontroller before exhausting the flash memory.

10.4.5   Engineering tierless IoT systems with mTask

A tierless Clean IoT system with microcontroller sensor nodes integrates a set of iTask tasks that specify the application and presentation layers with a set of mTasks that specify the perception and network layers. We illustrate with Clean WEMOS temperature sensor (CWTS): a simple room temperature sensor with a web display. CWTS is equivalent to the iTask CRTS (listing 10.3), except that the sensor node is a WEMOS microcontroller.

Listing 10.4 shows the complete program, and the key function is `devTask` with a top-level `Main` type (line 18). In mTask functions, shares, and devices can only be defined at this top level. The program uses the same shares `tempSDS` and `latestTemp` as CRTS, and for completeness we repeat those definitions. The body of `devTask` is the mTask slice of the program (lines 20 to 25). With `DHT` we again create a temperature sensor object `dht`. The iTask SDS `latestTemp` is first transformed to an SDS that accepts only temperature values, the `dateTimeStampedShare` adds the data via a lens. The `mapRead` adjusts the read type. This new SDS of type `Real` is lifted to the mTask program with `lowerSds`.

The `mainTask` is a simple iTask task that starts the `devTask` mTask task on the device identified by `deviceInfo` (line 29). At runtime the mTask slice is compiled to byte code, shipped to the indicated device, and launched. Thereafter, `mainTask` reads temperature values from the `latestTemp` SDS that is shared with the mTask device, and displays them on a web page (figure 10.5). The SDS—shared with the device using `lowerSds`—automatically communicates new temperature values from the microcontroller to the server.

While this simple application makes limited use of the mTask eDSL, it illustrates some powerful mTask program abstractions like basic tasks, task combinators, named recursive and parameterised tasks, and SDSs. Function composition (line 22) and currying (line 25) are inherited from the Clean host language. As mTask tasks are dynamically compiled, it is also possible to select and customise tasks as required at runtime. For example, the interval used in the `rpeatevery` task (line 23) could be a parameter to the `devTask` function.

```
1   module cwts
2
3   import mTask.Language, mTask.Interpret, mTask.Interpret.Device.TCP
4   import iTasks, iTasks.Extensions.DateTime
5
6   deviceInfo = {TCPSettings | host = "··· ", port = 8123, pingTimeout = ?None}   CO
7   interval = lit 10                                                              SN
8   DHT_pin = DigitalPin D4                                                        SI
9
10  Start world = doTasks mainTask world                                          WI
11
12  tempSDS :: SimpleSDSLens [(DateTime, Real)]
13  tempSDS = sharedStore "temperatures" []                                       DI
14
15  latestTemp :: SDSLens () (? (DateTime, Real)) (DateTime, Real)
16  latestTemp = mapReadWrite (listToMaybe, \x xs → ?Just [x:xs]) ?None
17          tempSDS                                                               DI
18  devTask :: Main (MTask v Real) | mtask, dht, lowerSds v
19  devTask =
20      DHT (DHT_DHT DHT_pin DHT11) \dht →                                        SI
21      lowerSds \localSds =                                                      CO
22          mapRead (snd o fromJust) (dateTimeStampedShare latestTemp)            SN
23      In {main = rpeatEvery (ExactSec interval)                                 SN
24          (temperature dht >>~.                                                 SI
25           setSds localSds)}                                                    SN
26
27  mainTask :: Task Real
28  mainTask
29      =   withDevice deviceInfo \dev → liftmTask devTask dev                    CO
30      -|| viewSharedInformation [] latestTemp                                   WI
31      <<@ Title "Latest temperature"                                           WI
```

Listing (Clean) 10.4: CWTS: a tierless temperature sensing IoT system. Written in Clean/iTask/mTask, it targets a resource-constrained sensor node. Each line is annotated with the functionality as analysed in section 10.6.1.
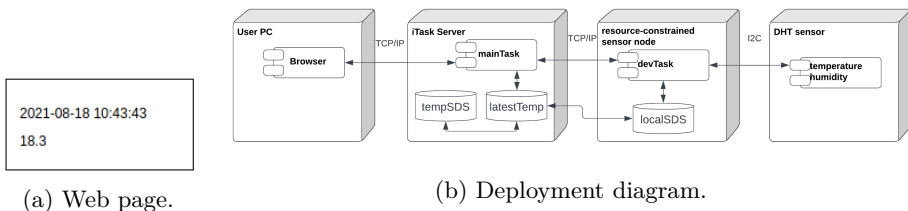

(a) Web page.

(b) Deployment diagram.

Figure 10.6: Tierless Clean/iTask/mTask CWTS temperature sensing IoT system.

10.5   UoG smart campus case study

The basis for our comparison between tiered and tierless technologies are four IoT
systems that all conform to the UoG smart campus specifications (section 10.5.3).
There is a small (12 room) deployment of the conventional Python-based PRS
stack that uses Raspberry Pi supersensors, and its direct comparator is the tierless
CRS implementation: also deployed on Raspberry Pis. To represent the more
common microcontroller sensor nodes we select ESP8266X powered WEMOS D1
Mini microcontrollers. To evaluate tierless technologies on microcontrollers we
compare the conventional Python/MicroPython PWS stack with the tierless CWS
implementation.

A similar range of commodity sensors is connected to both the Raspberry Pi
and WEMOS sensor nodes using various low-level communication protocols such as
GPIO, I^{2}C, SPI and 1-wire. The sensors are as follows: Temperature & Humidity:
LOLIN SHT30; Light: LOLIN BH1750; Motion: LOLIN PIR; Sound: SparkFun
SEN-12642; eCO$_2$: SparkFun CCS811.

Figure 10.7 shows both a prototype WEMOS-based sensor node and sensors
and a Raspberry Pi supersensor. Three different development teams developed the
four implementations: CWS and CRS were engineered by a single developer.
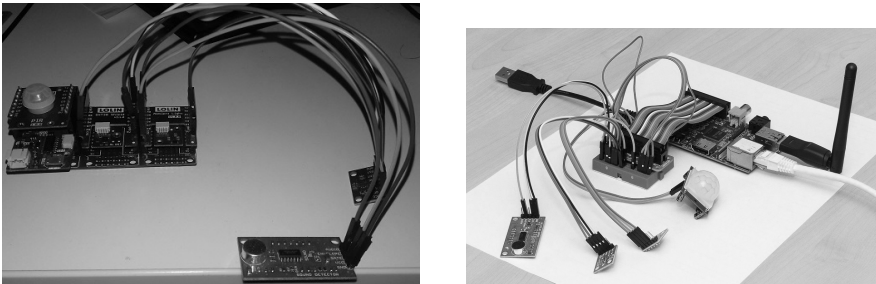
(a) A WEMOS used in PWS and CWS.   (b) A Raspberry Pi used in PRS and CRS.

Figure 10.7: Exposed views of sensor nodes.

10.5.1   Tiered implementations

The tiered PRS and PWS share the same server code executing on a commodity
PC (figure 10.1). The Python server stores incoming sensor data in two database
systems, i.e. Redis (in-memory data) and MongoDB (persistent data). The real-time
sensor data is made available via a streaming websockets server, which connects
with Redis. There is also an HTTP REST API for polling current and historical
sensor data, which hooks into MongoDB. Communication between a sensor node
and the server is always initiated by the node.

PRS's sensor nodes are relatively powerful Raspberry Pi 3 Model Bs. There is
a simple object-oriented Python collector for configuring the sensors and reading
their values. The collector daemon service marshals the sensor data and transmits

using MQTT to the central monitoring server at a preset frequency. The collector caches sensor data locally when the server is unreachable.

In contrast to PRS, PWS's sensor nodes are microcontrollers running MicroPython, a dialect of Python specifically designed to run on small, low powered embedded devices (Kodali and Mahesh, 2016). To enable a fair comparison between the software stacks we are careful to use the same object-oriented software architecture, e.g. using the same classes in PWS and PRS.

Python and MicroPython are appropriate tiered comparison languages. Tiered IoT systems are implemented in a whole range of programming languages, with Python, MicroPython, C and C++ being popular for some tiers in many implementations. C/C++ implementations would probably result in more verbose programs and even less type safety. The other reasons for selecting Python and MicroPython are pragmatic. PRS had been constructed in Python, deployed, and was being used as an IoT experimental platform. Selecting MicroPython for the resource-constrained PWS sensor nodes facilitates comparison by minimising changes to the resource-rich and resource-constrained codebases. We anticipate that the codebase for a tiered smart campus implementation in another imperative/object-oriented language, like C++, would be broadly similar to the PRS and PWS codebases.

10.5.2   Tierless implementations

The tierless CRS and CWS servers share the same iTask server code (figure 10.1), and can be compiled for many standard platforms. They use SQLite as a database backend. Communication between a sensor node and the server is initiated by the server.

CRS's sensor nodes are Raspberry Pi 4s, and execute Clean/iTask programs. Communication from the sensor node to the server is implicit and happens via SDSs over TCP using platform independent execution graph serialisation (Oortgiese et al., 2017).

CWS's sensor nodes are WEMOS microcontrollers running mTask tasks. Communication and serialisation is, by design, very similar to iTask, i.e. via SDSs over either a serial port connection, raw TCP, or MQTT over TCP.

Room m.01.07:	2020-04-30 09:15:03	
	Temperature:	20.4
	Humidity:	75
	Noise:	15
	Light:	70
	Motion:	✓
	Co2:	460

Temperature: 24.69

Humidity: 76.42

Sound Level: 51

Light: 223.9

CO2: 319

Motion: Active

Timestamp: 25 Aug 2021 06:53:27

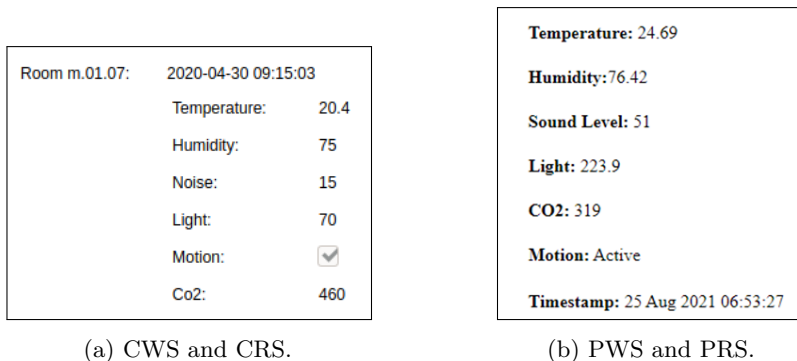(a) CWS and CRS.   (b) PWS and PRS.

Figure 10.8: Web interfaces for the smart campus application.

10.5.3   Operational equivalence

To ensure that the comparison reported in the following sections is based on IoT
stacks with equivalent functionality, we demonstrate that PWS, CWS and CRS, like
PRS, meet the functional requirements for the UoG smart campus sensor system.
We also compare the sensor node power consumption and memory footprint.

Functional requirements

The main goal of the UoG smart campus project is to provide a testbed for sensor
nodes and potentially other devices to act as a data collection and computation
platform for the UoG smart campus. The high-level functional requirements, as
specified by the UoG smart campus project board, are as follows. The system
should:

1. be able to measure temperature and humidity as well as light intensity,
2. scale to no more than 10 sensors per sensor node and investigate further
   sensor options like measuring sound levels,
3. have access to communication channels like Wi-Fi, Bluetooth and even wired
   networks.
4. have a centralised database server,
5. have a client interface to access information stored in the database,
6. provide some means of security and authentication,
7. have some means of managing and monitoring sensor nodes like updating
   software or detecting new sensor nodes.

All four smart campus implementations meet these high-level requirements.

Functional equivalence

Observation of the four implementations shows that they operate as expected, e.g.
detecting light or motion. To illustrate figure 10.8 shows the web interface for the
implementations where CWS and CRS are deployed in a different room from PWS
and PRS.

   All four implementations use an identical set of inexpensive sensors, so we expect
the accuracy of the data collected is within tolerance levels. This is validated by
comparing PRS and PWS sensor nodes deployed in the same room for some minutes.
The measurements show only small variances, e.g. temperatures recorded differ by
less than *0.4* °C, and light by less than *1* lux. For this room monitoring application
precise timings are not critical, and we don't compare the timing behaviours of the
implementations.

Memory and power consumption

Memory   By design sensor nodes are devices with limited computational capacity,
and memory is a key restriction. Even supersensors often have less than a GiB of
memory, and microcontrollers often have just tens of KiB. As the tierless languages

Table 10.1: UoG smart campus sensor nodes: maximum memory residency (in bytes).

PWS	PRS	CWS	CRS
20 270	3 557 806	880	2 726 680

synthesise the code to be executed on the sensor nodes, we need to confirm that the generated code is sufficiently memory efficient.

Table 10.1 shows the maximum memory residency after garbage collection of the sensor node for all four smart campus implementations. The smart campus sensor node programs executing on the WEMOS microcontrollers have low maximum residencies: *20 270* B for PWS and *880* B for CWS. In CWS the mTask system generates very high level TOP byte code that is interpreted by the mTask virtual machine and uses a small and predictable amount of heap memory. In PWS, the hand-written MicroPython is compiled to byte code for execution on the virtual machine. Low residency is achieved with a fixed size heap and efficient memory management. For example both MicroPython and mTask use fixed size allocation units and mark&sweep garbage collection to minimise memory usage at the cost of some execution time (Plamauer and Langer, 2017).

The smart campus sensor node programs executing on the Raspberry Pis have far higher maximum residencies than those executing on the microcontrollers: *3.5* MiB for PRS and *2.7* MiB for CRS. In CRS the sensor node code is a set of iTask executing on a full-fledged iTask server running in distributed child mode and this consumes far more memory. In PRS the sensor node program is written in Python, a language far less focused on minimising memory usage than MicroPython. For example an object like a string is larger in Python than in MicroPython and consequently does not support all features such as *f-strings*. Furthermore, not all advanced Python feature regarding classes are available in MicroPython, i.e. only a subset of the Python specification is supported (MicropythonTeam, 2022).

In summary the sensor node code generated by both tierless languages, iTask and mTask, is sufficiently memory efficient for the target sensor node hardware. Indeed, the maximum residencies of the Clean sensor node code is less than the corresponding hand-written (Micro)Python code. Of course in a tiered stack the hand-written code can be more easily optimised to minimise residency, and this could even entail using a memory efficient language like C/C++. However, such optimisation requires additional developer effort, and a new language would introduce additional semantic friction.

Power   Sensor nodes and sensors are designed to have low power demands, and this is particularly important if they are operating on batteries. The grey literature consensus is that with all sensors enabled a sensor node should typically have sub-*1* W peak power draw. The WEMOS sensor nodes used in CWS and PWS have the low power consumption of a typical embedded device: with all sensors enabled, they consume around *0.2* W. The Raspberry Pi supersensor node used

in CRS and PRS use more power as they have a general purpose ARM processor and run mainstream Linux. With all sensors enabled, they consume *1* W to *2* W, depending on ambient load. So a microcontroller sensor node consumes an order of magnitude less power than a supersensor node.

10.6 Is tierless IoT programming easier than tiered?

This section investigates whether tierless languages make IoT programming *easier* by comparing the UoG smart campus implementations. The CRS and CWS implementations allow us to evaluate tierless languages for resource-rich and for resource-constrained sensor nodes respectively. The PRS and PWS allow a like-for-like comparison with tiered Python implementations.

10.6.1　Comparing tiered and tierless codebases

Code size　is widely recognised as an approximate measure of the development and maintenance effort required for a software system (Rosenberg, 1997). SLOC is a common code size metric, and is especially useful for multi-paradigm systems like IoT systems. It is based on the simple principle that the more SLOC, the more developer effort and the increased likelihood of bugs (Rosenberg, 1997). It is a simple measure, not dependent on some formula, and can be automatically computed (Sheetz et al., 2009).

Of course SLOC must be used carefully as it is easily influenced by programming style, language paradigm, and counting method (Alpernas et al., 2020). Here we are counting lines of code to compare development effort, use the same idiomatic programming style in each component, and only count lines of code, omitting comments and blank lines.

Table 10.2 enumerates the SLOC required to implement the UoG smart campus functionalities in PWS, PRS, CWS and CRS. Both Python and Clean implementations use the same server and communication code for Raspberry Pi and for WEMOS sensor nodes (rows 5–7 of the table). The Sensor Interface (SI) refers to code facilitating the communication between the peripherals and the sensor node software. Sensor Node (SN) code contains all other code on the sensor node that does not belong to any another category, such as control flow. Manage Nodes (MN) is code that coordinates sensor nodes, e.g. to add a new sensor node to the system. Web Interface (WI) code provides the web interface from the server, i.e. the presentation layer. Database Interface (DI) code communicates between the server and the database(s). Communication (CO) code provides communication between the server and the sensor nodes, and executes on both sensor node and server, i.e. the network layer.

The most striking information in table 10.2 is that *the tierless implementations require far less code than the tiered implementations.* For example 166/562 SLOC for CWS/PWS, or 70% fewer SLOC. We attribute the code reduction to three factors: reduced interoperation, automatic communication, and high level programming abstractions. We analyse each of these aspects in the following subsections.

Table 10.2: Comparing tiered and tierless smart campus code sizes: SLOC and number of source files. PWS and CWS execute on resource-constrained sensor nodes, while PRS and CRS execute on resource-rich sensor nodes.

Code location	Functionality	Tiered Python		Tierless Clean	
		PWS	PRS	CWS	CRS
Sensor Node	Sensor Interface	52	57	11	11
	Sensor Node	178	183	9	4
Server	Manage Nodes	76		35	30
	Web Interface	56		28	
	Database Interface	106		78	
Communication	Communication	94	98	5	4
Total SLOC		562	576	166	155
№ Files		35	38	3	3

Code proportions. Comparing the percentages of code required to implement the smart campus functionalities normalises the data and avoids some issues when comparing SLOC for different programming languages, and especially for languages with different paradigms like object-oriented Python and functional Clean. Figure 10.9 shows the percentage of the total SLOC required to implement the smart campus functionalities in each of the four implementations, and is computed from the data in table 10.2. It shows that there are significant differences between the percentage of code for each functionality between the tiered and tierless implementations. For example 17% of the tiered implementations specifies communication, whereas this requires only 3% of the tierless implementations, i.e. $6\times$ less. We explore the reasons for this in section 10.6.4. The other major difference is the massive percentage of Database Interface code in the tierless implementations: at least 47%. The smart campus specification required a standard DBMS, and the Clean/iTask SQL interface occupies some 78 SLOC. While this is a little less than the 106 SLOC used in Python (table 10.2), it is a far higher percentage of systems with total codebases of only around 160 SLOC. Idiomatic Clean/iTask would use high level abstractions to store persistent data in an SDS, requiring just a few SLOC. The total size of CWS and CRS would be reduced by a factor of two and the percentage of Database Interface code would be even less than in the tiered Python implementations.

10.6.2 Comparing codebases for resource-rich/constrained sensor nodes

Before exploring the reasons for the smaller tierless codebase we compare the implementations for resource-rich and resource-constrained sensor nodes, again using SLOC and code proportions. Table 10.2 shows that the two tiered implementations are very similar in size: with PWS for microcontrollers requiring 562 SLOC and
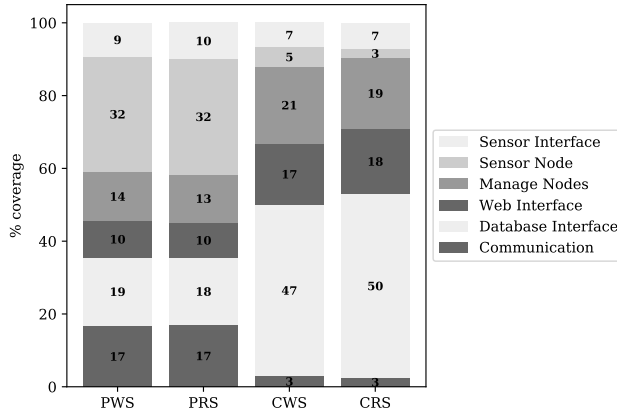
Figure 10.9: Comparing the percentage of code required to implement each functionality in tiered/tierless and resource-rich/constrained smart campus implementations.

PRS for supersensors requiring 576 SLOC. The two tierless implementations are also similar in size: CWS requiring 166 and CRS 155 SLOC.

There are several main reasons for the similarity. One is that the server-side code, i.e. for the presentation and application layers, is identical for both resource rich/constrained implementations. The identical server code accounts for approximately 40% of the PWS and PRS codebases, and approximately 85% of the CWS and CRS codebases (figure 10.9). For the perception and network layers on the sensor nodes, the Python and MicroPython implementations have the same structure, e.g. a class for each type of sensor, and use analogous libraries. Indeed, approaches like CircuitPython (CircuitPython Team, 2022) allow the same code to execute on both resource-rich and resource-constrained sensor nodes.

Like Python and MicroPython, iTask and mTask are designed to be similar, as elaborated in section 10.8. The similarity is apparent when comparing the iTask CRTS and Clean/iTask/mTask CWTS room temperature systems in listings 10.3 and 10.4. That is, both implementations use similar SDSs and lenses; they have similar `devTask`s that execute on the sensor node, and the server-side `mainTask`s are almost identical: they deploy the remote `devTask` before generating the web page to report the readings.

In both Python and Clean the resource-constrained implementations are less than 7% larger than the resource-rich implementations. This suggests that *the development and maintenance effort of simple IoT systems for resource-constrained and for resource-rich sensor nodes is similar in tierless technologies,* just as it is in tiered technologies. A caveat is that the smart campus system is relatively simple, and developing more complex perception and network code on bare metal may prove more challenging. That is, the lack of OS support, and the restricted

Table 10.3: Smart campus implementation languages comparison.

Code Location	Functionality	Languages			
		PWS	PRS	CWS	CRS
Sensor Node	Sensor Int.	MicroPython	Python	mTask	iTask
	Sensor Node	MicroPython	Python	mTask	iTask
Server	Manage Nodes	Python, JSON		iTask	
	Web Int.	HTML, PHP		iTask	
	Database Int.	Python,JSON,Redis		iTask	
Communication	Communication	MicroPython	Python	iTask,mTask	iTask
	Total	7	6	2	1

Table 10.4: Smart campus paradigm comparison.

Code Location	Functionality	Paradigms	
		Python	Clean
Sensor Node	Sensor Interface	Imperative	Declarative
	Sensor Node	Imperative	Declarative
Server	Manage Nodes	Imperative	Declarative
	Web Interface	Both	Declarative
	Database Interface	Both	Declarative
Communication	Communication	Imperative	Declarative
	Total	2	1

languages and libraries, may have greater impact. We return to this issue in section 10.8.

10.6.3 Reduced interoperation

The vast majority of IoT systems are implemented using a number of different programming languages and paradigms, and these must be effectively used and interoperated. A major reason that the tierless IoT implementations are simpler and shorter than the tiered implementations is that they use far fewer programming languages and paradigms. Here we use language to distinguish eDSLs from their host language: so iTask and mTask are considered distinct from Clean; and to distinguish dialects: so MicroPython is considered distinct from Python.

The tierless implementations use just two conceptually-similar DSLs embedded in the same host language, and a single paradigm (tables 10.3 and 10.4). In contrast, the tiers in PRS and PWS use six or more very different languages, and both imperative and declarative paradigms. Multiple languages are commonly used in other typical software systems like web stacks, e.g. a recent survey of open source projects reveals that on average at least five different languages are used (Mayer

and Bauer, 2015). Interoperating components in multiple languages and paradigms raises a plethora of issues.

Interoperation *increases the cognitive load on the developer* who must simultaneously think in multiple languages and paradigms. This is commonly known as semantic friction or impedance mismatch (Ireland et al., 2009). A simple illustration of this is that the tiered PRS source code comprises some 38 source and configuration files, whereas the tierless CRS requires just 3 files (table 10.2). The source could be structured as a single file, but to separate concerns is structured into three modules, one each for SDSs, types, and control logic (Stutterheim et al., 2018).

The developer must *correctly interoperate the components*, e.g. adhere to the API or communication protocols between components. The interoperation often entails additional programming tasks like marshalling or unmarshalling data between components. For example, in the tiered PRS and PWS architectures, JSON is used to serialise and deserialise data strings from the Python collector component before storing the data in the Redis database (listing 10.5).

```python
channel = 'sensor_status.%s.%s' % (hostname,
    sensor_types.sensor_type_name(s.sensor_type))
        self.r.publish(channel, s.SerializeToString())

.............................................................................

for message in p.listen():
    if message['type'] not in ['message', 'pmessage']:
        continue

try:
    status = collector_pb2.SensorStatus.FromString(message['data'])
```
Listing (Python) 10.5: JSON Data marshalling in PRS and PWS: sensor node above, server below.

To ensure correctness the developer *must maintain type safety* across a range of very different languages and diverse type systems, and we explore this further in section 10.7.1. The developer must also deal with the *potentially diverse failure modes*, not only of each component, but also of their interoperation, e.g. if a value of an unexpected type is passed through an API. We explore this further in section 10.7.2.

10.6.4   Automatic communication

In conventional tiered IoT implementations the developer must write and maintain code to communicate between tiers. For example PRS and PWS create, send and read MQTT (Light, 2017) messages between the perception and application layers. Table 10.2 shows that communication between these layers require some 94 SLOC in PWS and 98 in PRS, accounting for 17% of the codebase (bottom bars in figure 10.9). To illustrate, listing 10.6 shows part of the code to communicate sensor readings from the PWS sensor node to the Redis store on the server.

Not only must the tiered developer write additional code, but IoT communication code is often intricate. In such a distributed system the sender and receiver must be correctly configured, correctly follow the communication protocol through all execution states, and deal with potential failures. For example line 3 of listing 10.6: `redis host = config.get(`*`'Redis'`*`, `*`'Host'`*`)` will fail if either the host or IP are incorrect.

```python
def main():
    config.init('mqtt')
    redis_host = config.get('Redis', 'Host')
    redis_port = config.getint('Redis', 'Port')
    r = redis.StrictRedis(host=redis_host, port=redis_port)
    p = r.pubsub()
    p.psubscribe("sensor_status.*")
    for message in p.listen():
        if message['type'] not in ['message', 'pmessage']:
            print "Ignoring message %s" % message
            ...
```

Listing (Python) 10.6: Tiered communication example: MQTT transmission of sensor values in PWS.

In contrast, the tierless CWS and CRS communication is not only highly automated, but also automatically correct because matching sender and receiver code is generated by the compiler. Table 10.2 shows that communication is specified in just 5 SLOC in CWS and 4 in CRS, or just 3% of the codebase (bottom bars in figure 10.9).

Listing 10.4 illustrates communication in a tierless IoT language. That is, the CWTS temperature sensor requires just three lines of communication code, and uses just three communication functions. The `withDevice` function on line 29 integrates a sensor node with the server, allowing tasks to be sent to it. The `liftmTask` on line 29 integrates an mTask in the iTask runtime by compiling it and sending it for interpretation to the sensor node. The `lowerSds` on line 21 integrates SDSs from iTask into mTask, allowing mTask tasks to interact with data from the iTask server. The exchange of data, user interface, and communication are all automatically generated.

10.6.5 High level abstractions

Another reason that the tierless Clean implementations are concise is because they use powerful higher order IoT programming abstractions. For comprehensibility the simple temperature sensor from section 10.4.4 (listing 10.4) is used to compare the expressive power of Clean and Python-based IoT programming abstractions. There are implementations for all four configurations: Python Raspberry Pi temperature sensor (PRTS),[1] MicroPython WEMOS temperature sensor (PWTS),[1] CRTS[2] and

---

[1]Lubbers, M.; Koopman, P.; Ramsingh, A.; Singer, J.; Trinder, P. (2021): Source code, line counts and memory stats for PRS, PWS, PRT and PWT. Zenodo. 10.5281/zenodo.5081386.

Table 10.5: Comparing Clean and Python programming abstractions using the PWTS and CWTS temperature sensors (SLOC and total number of files).

Location	Functionality	PWTS	CWTS	Lines (listing 10.4)
Sensor Node	Sensor Interface	14	3	8, 20 and 24
	Sensor Node	67	4	7, 22, 23 and 25
Server	Web Interface	17	3	10, 30 and 31
	Database Interface	106	2	13 and 16
Communication	Communication	94	3	6, 21 and 29
Total SLOC		298	15	
№ Files		27	1	

CWTS.[2] but as the programming abstractions are broadly similar, we compare only the PWTS and CWTS implementations.

Although the temperature sensor applications are small compared to the smart campus application, they share some typical IoT stack traits. The architecture consists of a server and a single sensor node (figure 10.6). The sensor node measures and reports the temperature every ten seconds to the server while the server displays the latest temperature via a web interface to the user.

Table 10.5 compares the SLOC required for the MicroPython and Clean/iTask/ mTask WEMOS temperature sensors: PWTS and CWTS respectively. The code sizes here should not be used to compare the programming models as implementing such a small application as a conventional IoT stack requires a significant amount of configuration and other machinery that would be reused in a larger application. Hence, the ratio between total PWTS and CWTS code sizes (298:15) is far greater than for realistic applications like PWS and CWS (471:166).

The multiple tiers in PRS and PWS provide different levels of abstraction and separation of concerns. However, there are various ways that high-level abstractions make the CWS much shorter than PRS and PWS implementations.

Firstly, FP languages are generally more concise than most other programming languages because their powerful abstractions like higher-order and/or polymorphic functions require less code to describe a computation. Secondly, the TOP paradigm used in iTask and mTask reduces the code size further by making it easy to specify IoT functionality concisely. As examples, the step combinator >>*. allows the task value on the left-hand side to be observed until one of the steps is enabled; and the `viewSharedInformation` (line 31 of listing 10.4) part of the UI will be automatically updated when the value of the SDS changes. Moreover, each SDS provides automatic updates to all coupled SDSs and associated tasks. Thirdly, the amount of explicit type information is minimised in comparison to other languages, as much is automatically inferred (Hughes, 1989).

---

[2]Lubbers, M.; Koopman, P.; Ramsingh, A.; Singer, J.; Trinder, P. (2021): Source code, line counts and memory stats for CRS, CWS, CRTS and CWTS. Zenodo. 10.5281/zenodo.5040754.

10.7 Could tierless IoT programming be more reliable than tiered?

This section investigates whether tierless languages make IoT programming more reliable. Arguably the much smaller and simpler code base is inherently more understandable, and more likely to be correct. Here we explore specific language issues, namely those of preserving type safety, maintainability, failure management, and community support.

10.7.1 Type safety

Strong typing identifies errors early in the development cycle, and hence plays a crucial role in improving software quality. In consequence almost all modern languages provide strong typing, and encourage static typing to minimise runtime errors. That said, many distributed system components written in languages that primarily use static typing, like Haskell and Scala, use some dynamic typing, e.g. to ensure that the data arriving in a message has the anticipated type (Epstein et al., 2011; Gupta, 2012).

In a typical tiered multi-language IoT system the developer must integrate software in different languages with very different type systems, and potentially executing on different hardware. The challenges of maintaining type safety have long been recognised as a major component of the semantic friction in multi-language systems, e.g. Ireland et al. (2009).

Even if the different languages used in two components are both strongly typed, they may attribute, often quite subtly, different types to a value. Such type errors can lead to runtime errors, or the application silently reporting erroneous data. Such errors can be hard to find. Automatic detection of such errors is sometimes possible, but requires an addition tool like Jinn (Furr and Foster, 2005; Lee et al., 2010).

```
message SensorData {
    enum SensorType { TEMPERATURE = 1; ... }
    SensorType sensor_type   = 1;
    uint64     timestamp      = 2;
    double     float_value    = 3;
}
.............................................................................
channel = 'sensor_status.%s.%s' % (hostname,
    sensor_types.sensor_type_name(s.sensor_type))
        self.r.publish(channel, s.SerializeToString())
```

Listing (Python) 10.7: PRS loses type safety as a sensor node sends a `double`, and the server stores a `string`.

Analysis of the PRS codebase reveals an instance where it, fairly innocuously, loses type safety. The fragment in listing 10.7 first shows a `double` sensor value being sent from the sensor node, and then shows the value being stored in Redis

as a `string` on the server. As PWS preserves the same server components it also suffers from the same loss of type safety.

A tierless language makes it possible to guarantee type safety across an entire IoT stack. For example the Clean compiler guarantees static type safety as the entire CWS software stack is type checked, and generated, from a single source. Tierless web stack languages like Links (Cooper et al., 2007) and Hop (Serrano et al., 2006) provide the same guarantee for web stacks.

10.7.2   Failure management

Some IoT applications, including smart campus and other building monitoring applications, require high sensor uptimes. Hence, if a sensor or sensor node fails the application layer must be notified, so that it can report the failure. In the UoG smart campus system a building manager is alerted to replace the failed device.

In many IoT architectures, including PRS and PWS, detecting failure is challenging because the application layer listens to the devices. When a device comes online, it registered with the application and starts sending data. When a device goes offline again, it could be because the power was out, the device was broken or the device just paused the connection.

If a sensor node fails in CWS, the iTask/mTask combinator interacting with a sensor node will throw an iTask exception. The exception is propagated and a handler can respond, e.g. rescheduling the task on a different device in the room, or requesting that a manager replaces the device. That is, iTask, uses standard succinct declarative exception handling.

```
failover :: [TCPSettings] (Main (MTask BCInterpret a)) → Task a
failover []     _     = throw "Exhausted device pool"
failover [d:ds] mtask = try (withDevice d (liftmTask mtask)) except
where except MTEUnexpectedDisconnect = failover ds mtask
      except e                       = throw e
```

Listing (Clean) 10.8: An mTask failover combinator.

In the UoG smart campus application, this can be done by creating a pool of sensor nodes for each room and when a sensor node fails, assign another one to the task. Listing 10.8 shows a failover combinator that executes an mTask on one of a pool of sensor nodes. If a sensor node unexpectedly disconnects, the next sensor node is tried until there are no sensor nodes left. If other errors occur they are propagated as usual.

Currently, PRS and PWS both use heartbeats to confirm that the sensor nodes are operational, and will report failures. At the cost of extending the codebase, failover to an alternate sensor node could be provided.

10.7.3   Maintainability

Far more engineering effort is expended on maintaining a system, than on the initial development. Tiered and tierless IoT systems have very different maintainability properties.

The modularity of the tiered stack makes replacing tiers/components easy. For example in PWS or PRS the MongoDB NoSQL DBMS could be readily be replaced by an alternative like CouchDB. Because a tierless compiler must generate code for components, replacing them may not be so easy. If there are iTask abstractions for the component then replacement is straightforward. For example replacing SQLite with some other SQL DBMS simply entails recompilation of the application. However incorporating a component that does not yet have a task abstraction, like a NoSQL DBMS, is more involved. That is, a foreign function interface to the new component must be implemented, along with a suitable iTask abstraction for operations on the component.

Many maintenance tasks are smaller in scale and occur within the components or tiers. Consider a simple change, for example if the temperature value recorded by a sensor changes from integer to real.

All tiers of a tiered stack must be correctly and consistently refactored to reflect the change of temperature data type: so changes at the perception, network, application and presentation layers. A PWS developer works in seven languages and two paradigms to effect the change (table 10.2), and must edit many source files. Many programming errors are either detected at runtime when testing the stack, or worse not automatically detected and produce erroneous results.

In a tierless language the source code is much smaller and so it is easier to comprehend, i.e. to understand what refactoring is required. A CWS developer works in only two languages and a single paradigm to effect the change, and will edit no more than three source files (table 10.2). Moreover, the compiler will statically detect many programming errors.

More substantial in-component maintenance raises similar issues as for tiered implementations. If the maintenance activity requires a new task combinator, this is readily constructed in iTask, but may require changing the DSL implementation in mTask, i.e. to change the compiler and the byte code interpreter. That is, mTask is more *brittle* than iTask.

In summary, while a tiered approach makes replacing components easy, refactoring within the components is far harder in a multi-tier multi-language IoT implementation than in a tierless IoT implementation.

10.7.4   Support

Community and tool support are essential for engineering reliable production software. PRS and PWS are both Python based, and Python/MicroPython are among the most popular programming languages (Cass, 2020). Python is also a common choice for some tiers of IoT applications (Tanganelli et al., 2015). Hence, there are a wide range of development tools like IDEs and debuggers, a thriving community and a wealth of training material. There are even specialised IoT Boards like PyBoard & WiPy that are specifically programmed using Python variations like MicroPython.

In contrast, tierless languages are far less mature than the languages used in tiered stacks, and far less widely adopted. This means that for CWS and CRS there are fewer tools, a far smaller developer community, and less training material available.

CWS and CRS are both written in DSLs embedded in Clean, a fairly stable industrial-grade but niche FP language. The DSLs are implemented in Clean but require experimental compiler extensions that are often undocumented. There are few maintainers of the DSLs and documentation is often sparse. Acquiring information about the systems requires distilling academic papers and referring to the source code. There is a Clean IDE, but it does not contain support for the iTask or mTask DSLs.

10.8 Comparing tierless languages for resource-rich/constrained sensor nodes

This section compares two tierless IoT languages: one for resource-rich, and the other for resource-constrained, sensor nodes. Key issues are the extent to which the very significant resource constraints of a microcontroller limit the language, and the benefits of executing on bare metal, i.e. without an OS.

With the tierless Clean technologies described here, iTask are always used to program the application and presentation layers of the IoT stack. So any differences occur in the perception and network layer programming. If sensor nodes have the capacity to support iTask, a tierless IoT system can be constructed in Clean using only iTask, as in CRS. Alternatively for sensor nodes with low computational power, like typical microcontrollers, mTask is used for the perception and network layers, as in CWS. This section compares the iTask and mTask eDSLs, with reference to CRS and CWS as exemplars. Table 10.6 summarises the differences between the Clean embedded IoT eDSLs and their host language.

10.8.1 Language restrictions for resource-constrained execution

Executing components on a resource-constrained sensor node imposes restrictions on programming abstractions available in a tierless IoT language or DSL. The small and fixed-size memory are key limitations. The limitations are shared by any high-level language that targets microcontrollers such as BIT, PICBIT, PICOBIT, Microscheme and uLisp (St-Amour and Feeley, 2009; Dubé, 2000; Feeley and Dubé, 2003; Johnson-Davies, 2020; Suchocki and Kalvala, 2015). Even in low level languages some language features are disabled by default when targeting microcontrollers, such as runtime type information (RTTI) in C++.

Here we investigate the restrictions imposed by resource-constrained sensor nodes on mTask, in comparison with iTask. While iTask and mTask are by design superficially similar languages, to execute on resource-constrained sensor nodes mTask tasks are more restricted, and have a different semantics.

Table 10.6: Comparing tierless IoT DSLs for resource-rich sensor nodes (iTask), for resource-constrained sensor nodes (mTask), and their Clean host language.

Property	Clean	iTask	mTask
Function for an IoT System	Host Language	Specify distributed workflows	Specify sensor node workflow
Referentially transparent	Yes	Yes	Yes
Evaluation strategy	Lazy	Lazy	Strict
Higher-order functions	Yes	Yes	No
User-defined datatypes	Yes	Yes	No
Task oriented	No	Yes	Yes
Higher-order tasks	–	Yes	No
Execution Target	Commodity PC	Commodity PC and browser	Microcontroller Microcontroller
Language Implementation	Compiled or interpreted	Compiled and interpreted	Interpreted

Programs in mTask do not support user defined higher order functions, the only higher order functions available are the predefined mTask combinators. Programmers can, however, use any construct of the Clean host language to construct an mTask program, including higher order functions and arbitrary data types. For example folding an mTask combinator over a list of tasks. The only restriction is that any higher order function must be macro expanded to a first order mTask program before being compiled to byte code. As an example in listing 10.4 we use `temperature dht >>~. setSds localSds` instead of `temperature dht >>~. \temp →  setSds localSds temp`.

In contrast to iTask, mTask programs have no user defined or recursive data types. It is possible to add user defined types—as long as they are not sum types—to mTask, but this requires significant programming effort. Due to the language being shallowly embedded, pattern matching and field selection on user defined types is not readily available and thus needs to be built into the language by hand. Alleviating this limitation remains future work.

Programs in mTask mainly use strict rather than lazy evaluation to minimise the requirement for a variable size heap. This has no significant impact for the mTask programs we have developed here, nor in other IoT applications we have engineered.

Abstractions in mTask are less easily extended than iTask. For example iTask can be extended with a new combinator that composes a specific set of tasks for some application. Without higher order functions the equivalent combinator can often not be expressed in mTask, and adding it to mTask requires extending the DSL rather than writing a new definition in it. On the other hand, it is possible to outsource this logic to the iTask program as mTask and iTask tasks are so tightly integrated.

10.8.2 The benefits of a bare metal execution environment

Despite the language restrictions, components of a tierless language executing on a microcontroller can exploit the bare metal environment. Many of these benefits are shared by other bare metal languages like MicroPython or C/C++. So as mTask executes on bare metal it has some advantages over iTask. Most notably mTask has better control of timing as on bare metal there are no other processes or threads that compete for CPU cycles. This makes the mTask `repeatEvery` (listing 10.4, line 23) much more accurate than the iTask `waitForTimer` (listing 10.3, line 13). While exact timing is not important in this example, it is significant for many other IoT applications. In contrast iTask cannot give real time guarantees. One reason is that an iTask server can ship an arbitrary number of iTask or mTask tasks to a device. Such competing tasks, or indeed other OS threads and processes, consume processor time and reduce the accuracy of timings. However, even when using multiple mTask tasks, it is easier to control the number of tasks on a device than controlling the number of processes and threads executing under an OS.

An mTask program has more control over energy consumption. The mTask eDSL and the mTask RTS are designed to minimise energy usage (Crooijmans, 2021). Intensional analysis of the declarative task description and current progress at run time allow the RTS to schedule tasks and maximise idle time. As the RTS is the only program running on the device, it can enforce deep sleep and wake up without having to worry about influencing other processes.

The mTask RTS has direct control of the peripherals attached to the microcontroller, e.g. over GPIO pins. There is no interaction with, or permission required from, the OS. Moreover, microcontrollers typically have better support for hardware interrupts, reducing the need to poll peripherals. The downside of this direct control is that CWS has to handle some exceptions that would otherwise be handled by the OS in CRS and hence the device management code is longer: 28 versus 20 SLOC in table 10.2.

10.8.3 Summary

Table 10.6 summarises the differences between the Clean IoT eDSL and their host language. The restrictions imposed by a resource-constrained execution environment on the tierless IoT language are relatively minor. Moreover the mTask programming abstraction is broadly compatible with iTask. As a simple example compare the iTask and mTask temperature sensors in listings 10.3 and 10.4. As a more realistic example, the mTask based CWS smart campus implementation is similar to the iTask based CRS, and requires less than 10% additional code: 166 SLOC compared with 155 SLOC (table 10.2).

Even with these restrictions, mTask programming is at a far higher level of abstraction than almost all bare metal languages, e.g. BIT, PICBIT, PICOBIT and Microscheme. That is mTask provides a set of higher order task combinators, shared distributed data stores, &c. (section 10.4.4). Moreover, it seems that common sensor node programs are readily expressed using mTask. In addition to the CWTS and CWS systems outlined here, other case studies include Arduino examples as well as some bigger tasks (Koopman et al., 2018; Lubbers et al., 2019; Lubbers

et al., 2023b). We conclude that the programming of sensor tasks is well-supported by both DSLs.

10.9   Conclusion

10.9.1   Summary

We have conducted a systematic comparative evaluation of two tierless language technologies for IoT stacks: one for resource-rich, and the other for resource-constrained sensor nodes. The basis is four implementations of a deployed smart campus IoT stack: two conventional tiered and Python-based stacks, and two tierless Clean stacks. An operational comparison of implementations demonstrates that they have equivalent functionality, and meet the UoG smart campus functional requirements (section 10.5).

We show that *tierless languages have the potential to significantly reduce the development effort for IoT systems*. Specifically the tierless CWS and CRS stacks require far less code, i.e. 70% fewer SLOC, than the tiered PWS and PRS stacks (table 10.2). We analyse the code reduction and attribute it to the following three main factors. 1. Tierless developers need to manage less interoperation: CRS uses a single DSL and paradigm, and CWS uses two DSLs in a single paradigm and three source code files. In contrast, both PRS and PWS use at least six languages in two paradigms and spread over at least 35 source code files (tables 10.2 to 10.4). Thus, a tierless stack minimises semantic friction. 2. Tierless developers benefit from automatically generated, and hence correct, communication (listing 10.4), and write 6× less communication code (figure 10.9). 3. Tierless developers can exploit powerful high-level declarative and task-oriented IoT programming abstractions (table 10.5), specifically the composable, higher-order task combinators outlined in section 10.4.2. Our empirical results for IoT systems are consistent with the benefits claimed for tierless languages in other application domains. Namely that a tierless language provides a *Higher Abstraction Level*, *Improved Software Design*, and improved *Program Comprehension* (Weisenburger et al., 2020).

We show that *tierless languages have the potential to significantly improve the reliability of IoT systems*. We illustrate how Clean maintains type safety, contrasting this with a loss of type safety in PRS. We illustrate higher order failure management in Clean/iTask/mTask in contrast to the Python-based failure management in PRS. For maintainability a tiered approach makes replacing components easy, but refactoring within the components is far harder than in a tierless IoT language. Again our findings are consistent with the simplified *Code Maintenance* benefits claimed for tierless languages (Weisenburger et al., 2020). Finally, we contrast community support for the technologies (section 10.7).

We report *the first comparison of a tierless IoT codebase for resource-rich sensor nodes with one for resource-constrained sensor nodes*. 1. The tierless implementations have very similar code sizes (SLOC), as do the tiered implementations: less than 7% difference in table 10.2. This suggests that the development and maintenance effort of simple tierless IoT systems for resource-constrained and for resource-rich sensor nodes is similar, as it is for tiered technologies. 2. The

percentages of code required to implement each IoT functionality in the tierless Clean implementations is very similar as it is in the tiered Python implementations (figure 10.9). This suggests that the code for resource-constrained and resource-rich sensor nodes can be broadly similar in tierless technologies, as it is in many tiered technologies (section 10.6.2).

We present *the first comparison of two tierless IoT languages: one designed for resource-constrained sensor nodes (Clean/iTask/mTask), and the other for resource-rich sensor nodes (Clean/iTask).* Clean/iTask can implement all layers of the IoT stack if the sensor nodes have the computational resources, as the Raspberry Pis do in CRS. On resource constrained sensor nodes mTask are required to implement the perception and network layers, as on the WEMOS minis in CWS. We show that a bare metal execution environment allows mTask to have better control of peripherals, timing and energy consumption. The memory available on a microcontroller restricts the programming abstractions available in mTask to a fixed set of combinators, no user defined or recursive data types, strict evaluation, and makes it harder to add new abstractions. Even with these restrictions mTask provide a higher level of abstraction than most bare metal languages, and can readily express many IoT applications including the CWS UoG smart campus application (section 10.8). Our empirical results are consistent with the benefits of tierless languages listed in Weisenburger et al. (2020, section 2.1).

10.9.2   Reflections

This study is based on a specific pair of tierless IoT languages, and the Clean language frameworks represent a specific set of tierless language design decisions. Many alternative tierless IoT language designs are possible, and some are outlined in section 10.3.3. Crucially the limitations of the tierless Clean languages, e.g. that they currently provide limited security, should not be seen as limitations of tierless technologies in general.

This study has explored some, but not all, of the potential benefits of tierless languages for IoT systems. An IoT system specified as a single tierless program is amenable to a host of programming language technologies. For example, if the language has a formal semantics, as Links, Hop and Clean tasks do (Cooper et al., 2007; Plasmeijer et al., 2012; Serrano et al., 2006), it is possible to prove properties of the system, e.g. (Steenvoorden et al., 2019). As another example program analyses can be applied, and section 10.3.3 and (Weisenburger et al., 2020) outline some of the analyses could be, and in some cases have been, used to improve IoT systems. Examples include automatic tier splitting (Philips et al., 2014), and controlling information flow to enhance security (Valliappan et al., 2020).

While offering real benefits for IoT systems development, tierless languages also raise some challenges. Programmers must master new tierless programming abstractions, and the semantics of these automatic multi-tier behaviours are necessarily relatively complex. In the Clean context this entails becoming proficient with the iTask and mTask DSLs. Moreover, specifying a behaviour that is not already provided by the tierless language requires either a workaround, or extending a DSL. However, implementing the relatively simple smart campus application required

no such adaption. Finally, tierless IoT technology is very new, and both tool and community support have yet to mature.

10.9.3 Future work

This chapter is a technology comparison between tiered and tierless technologies. The metrics reported, such as code size, numbers of source code files, and of paradigms are only indirect, although widely accepted, measures of development effort. A more convincing evaluation of tierless technologies could be provided by conducting a carefully designed and substantial user study, e.g. using N-version programming.

A study that implemented common benchmarks or a case study in multiple tierless IoT languages would provide additional evidence for the generality of the tierless approach. Such a study would enable the demonstration and comparison of alternative design decisions within tierless languages, as outlined in section 10.3.3.

In ongoing work we are extending the mTask system in various ways. One extension allows mTask tasks to communicate directly, rather than via the iTask server. Another provides better energy management, which is crucial for battery powered sensor nodes.

Acknowledgements

Chapter 11

Coda

This chapter concludes the dissertation and reflects on the work.

11.1   Reflections

This dissertation shed light on orchestrating complete IoT systems using TOP. The term IoT refers to the interconnected network of physical devices that are connected to each other and the internet. The edge, or perception, layer of an IoT system is often powered by microcontrollers. These small and cheap computers do not have powerful hardware but are energy efficient and support many sensors and actuators. While the term IoT has already been known for almost thirty years, only recently, the exponential growth of the number of IoT edge devices is really ramping up. Programming IoT systems is very complex because each layer of the system is built with different computers, hardware architectures, programming languages, programming paradigms, and abstraction levels. This generates a lot of semantic friction and interoperation issues. Furthermore, IoT systems become convoluted because they are dynamic, multi-tiered, multi-user, multitasking, interactive, distributed, and collaborative in nature. TOP proves a suitable programming paradigm that allows the declarative specification of exactly such systems. However, edge devices are often too computationally restricted to be able to run a full-fledged TOP system such as iTask. The dissertation is structured as a purely functional rhapsody in three episodes.

In order to get TOP to resource-constrained edge devices we use special tools: DSLs. The dissertation shows several techniques for creating eDSLs in episode I. Then it shows a tool, mTask, a TOP system for IoT edge devices in episode II. Finally, in episode III it compares how this approach compares to existing approaches for programming IoT systems.

11.1.1 Étude — Domain-Specific Languages

Episode I presents some tool crafting techniques that are useful for creating TOP languages for IoT edge devices. It presents two novel techniques for embedding DSLs in FP languages. Both technique make it easier for DSL developers to create rich and extensible DSLs.

Classy deep embedding is a novel eDSL embedding technique. When embedding DSLs, one always has to make concessions. It is either easy to add language constructs, or to add interpretations of the terms, but never both. Some advanced embedding techniques found ways of mitigate this issue. Tagless-final embedding offers a way of extending a shallowly embedded DSL both in constructs and interpretations. Classy deep embedding is the organically grown counterpart for deep embedding a DSL. It allows orthogonal extension of language constructs and interpretations with minimal boilerplate and no advanced type system extensions.

When embedding a DSL in a language, much, but not all, of the machinery is inherited. An example of this are host-language data types. They are not automatically useable in the DSL because the interfaces such as constructors, deconstructors, constructor predicates, and pattern matching are not inherited. I show how to automatically generate the required boilerplate for shallowly embedded DSLs in order to make data types from the host language first-class citizens in the DSL. The scaffolding is generated using template metaprogramming and quasiquotation is used to alleviate the programmer from the syntax burden and support pattern matching.

11.1.2 Orchestrating the Internet of Things using Task-Oriented Programming

General-purpose TOP systems cannot run on edge devices due to their significant hardware requirements. However, with the right techniques, DSLs can be created that can be executed on edge devices while maintaining the high abstraction level. By embedding domain-specific knowledge into the language and execution platform, and leaving out general-purpose functionality, TOP languages can be made suitable for edge devices.

Episode II contains a complete overview of such a tool: the mTask system. The mTask language is a unique domain-specific TOP eDSL designed system for edge devices. The mTask system is fully integrated with the iTask system, a TOP system for programming distributed web applications. In the iTask system, there are abstractions for details such as user interfaces, data storage, client-side platforms, and persistent workflows. The mTask language abstracts away from edge device specific details such as sensor and actuator access, heterogeneity in hardware, and multitasking and scheduling. Tasks in the mTask system are compiled at run time and sent to the device dynamically in order to support create dynamic systems where tasks are tailor-made for the current work requirements. Using only three simple functions, devices are connected to iTask servers, mTask tasks are integrated in iTask, and iTask SDSs accessed from within mTask tasks. Its design, integration with iTask, implementation, and green computing facilities are shown. This tight

integration makes programming full IoT systems using TOP possible without major compromises.

11.1.3 Tiered versus Tierless Programming

Using tierless programming, many issues that arise with tiered programming are mitigated. This has already been observed in web applications. The mTask system show that it is possible to program edge devices of a IoT systems using TOP. Furthermore, when used together with iTask, entire IoT systems can be programmed tierlessly. Whether this novel approach to programming tiered systems also reduces the IoT development grief is answered in episode III. This episode presents a four-way qualitative and quantitative comparison of the following systems: PRS, a tiered system based on resource-rich edge devices powered by Python; PWS, a tiered system based on resource-constrained edge devices by MicroPython; CRS, a tierless system based on resource-rich edge devices powered by iTask; CWS, a tierless system based on resource-constrained edge devices powered by mTask.

This comparison shows that when using a programming paradigm that is available both for resource-rich and resource-constrained edge devices, there is little difference in developer grief. On the other hand, using a tierless system compared to a tiered system reduces the developer grief significantly.

Every layer of the entire IoT system is specified in a single source, the same strong type system, and similar high abstraction level. The tierless approach results in fewer SLOC, files, programming languages and programming paradigms. All code is simultaneously checked by a single compiler, reducing interoperability problems. Furthermore, all communication and integration is automatically generated, reducing interoperability issues even more.

However, it is not a silver bullet, there are some disadvantages as well. Tierless languages are novel, and hence often lack tooling and community support. They contain high-level tierless abstractions that the programmer has to master. The low-level specific semantics of the final application may become more difficult to distill from the specification. Finally, the system is more monolithic compared to tiered approaches. Changing components within the system is easy if it already is supported in the eDSL, but adding new components to the system requires the programmer to add it to all complex components of the languages such as the compiler, and RTS.

Appendix A

Clean for Haskell programmers

This appendix is meant give people who are familiar with the FP language Haskell a concise overview of the programming language Clean and how it differs from Haskell. The goal is to support the reader when reading Clean code. Table A.1 shows frequently occurring Clean language elements on the left side and their Haskell equivalent on the right side. Obviously, this summary is not exhaustive. Some Clean language elements that are not easily translatable to Haskell and thus do not occur in the summary following below. We hope you enjoy these notes and that it aids you in reading Clean programs.

Clean—acronym for Clean Language of East-Anglia and Nijmegen (Barendregt et al., 1987)—, was originally designed as a graph rewriting system (GRS) core language but quickly served as an intermediate language for other functional languages (Brus et al., 1987). In the early days it has also been called *Concurrent* Clean (Nöcker et al., 1991) but these days the language has no support for concurrency anymore. Fast-forward thirty years, Clean is now a robust language with state-of-the-art features and is actually used in industry as well as academia—albeit in select areas of the world.

Initially, when it was used mostly as an intermediate language, it had a fairly spartan syntax. However, over the years, the syntax got friendlier and it currently it looks a lot like Haskell. In the past, a *double-edged* fronted even existed that allowed Clean to be extended with Haskell98 syntax and vice versa (van Groningen et al., 2010), however this frontend is no longer maintained. This chapter gives only a brief syntactical and functional comparison. A complete specification of the Clean language can be found in the latest language report (Plasmeijer et al., 2021). Much of this is based on work by Achten, although that was based on Clean 2.1 and Haskell98 (Achten, 2007). When Haskell is mentioned we actually mean GHC's Haskell[1] and by Clean we mean Clean 3.1's compiler with the iTask extensions.

---

[1] If an extension is enabled, a footnote is added stating that `SomeExtension` is required.

A.1   Features

A.1.1   Modules

Clean has separate implementation and definition modules. The definition module contains the class definitions, instances, function types and type definitions (possibly abstract). Implementation modules contain the function implementations as well. This means that only what is defined in the definition module is exported in Clean. This differs greatly from Haskell, as there is only a module file there. Choosing what is exported in Haskell is done using the **module** Mod(···) syntax.

A.1.2   Strictness

In Clean, by default, all expressions are evaluated lazily. Types can be annotated with a strictness attributes (!), resulting in the values being evaluated to head-normal form before the function is entered. In Haskell, in patterns, strictness is enforced using !.[2] Within functions, the strict let (#!) is used to force evaluate an expression, in Haskell seq or $! is used for this.

A.1.3   Uniqueness typing

Types in Clean may be *unique*, which means that instances of the type cannot be shared (Barendsen and Smetsers, 1996). The uniqueness type system allows the compiler to generate efficient code because unique data structures can be destructively updated. Furthermore, uniqueness typing serves as a model for side effects as well (Achten et al., 1993; Achten and Plasmeijer, 1995). Clean uses the *world-as-value* paradigm where World represents the external environment and is always unique (Backus et al., 1990). A program with side effects is characterised by a Start :: *World → *World start function. In Haskell, interaction with the world is done using the IO monad (Peyton Jones and Wadler, 1993). An IO monad could very well be—and actually is—implemented in Clean using a state monad with the World as a state. Besides marking types as unique, it is also possible to mark them with uniqueness attributes variables u: and define constraints on them. For example, to make sure that an argument of a function is at least as unique as another argument. Finally, using . (a full stop), it is possible to state that several variables are equally unique. Uniqueness is propagated automatically in function types but must be marked manually in data types. Examples can be seen in listing A.1.

```
f :: *a → *a               // f works on unique values only
f :: .a → .a               // f works on unique and non−unique values
f :: v:a u:b → u:b, [v<=u]  // f works when a is less unique than b
```
Listing (Clean) A.1: Examples of uniqueness annotations.

---

[2]Requires BangPatterns to be enabled.

A.1.4 Expressions

Patterns in Clean can be used as predicates as well (Plasmeijer et al., 2021, section 3.4.3). Using the `=:` operator, a value is tested against a pattern. Variable names are not allowed but wildcard patterns (`_`) are.

```clean
isNil :: [a] → Bool
isNil l = l=:[]


:: T = A Int | B Bool


ifAB :: T a a → a
ifAB x ifa ifb = if (x =: (A _)) ifa ifb
```

Listing (Clean) A.2: Examples of *matches pattern* expressions.

Due to the nature of uniqueness typing, many functions in Clean are state transition functions with possibly unique states. The *let before* construct allows the programmer to specify sequential actions without having to invent unique names for the different versions of the state. Listing A.3 shows an example of the usage of the *let before* construct (adapted from (Plasmeijer et al., 2021, section 3.5.4)).

```clean
readChars :: *File → ([Char], *File)
readChars file
# (ok, char, file) = freadc file
| not ok           = ([], file)
# (chars, file)    = readChars file
= ([char:chars], file)
```

Listing (Clean) A.3: Let before expression example.

A.1.5 Generics

Polytypic functions (Jeuring and Jansson, 1996)—also known as generic or kind-indexed functions—are built into Clean (Plasmeijer et al., 2021, section 7.1)(Alimarine, 2005) whereas in Haskell they are implemented as a library (GHC Team, 2021b, section 6.19.1). The syntax of the built-in generics of Clean is very similar to that of Generic H∀skell (Hinze and Jeuring, 2003).

For example, defining a generic equality is done as in listing A.4.

```clean
generic gEq a :: a a → Bool


gEq{|Int|}           x           y           = x == y
gEq{|Bool|}          x           y           = x == y
gEq{|Real|}          x           y           = x == y
gEq{|Char|}          x           y           = x == y
gEq{|UNIT|}          x           y           = True
gEq{|OBJECT|} f      (OBJECT x)  (OBJECT y)  = f x y
gEq{|CONS|}   f      (CONS x)    (CONS y)    = f x y
gEq{|RECORD|} f      (RECORD x)  (RECORD y)  = f x y
gEq{|FIELD|}  f      (FIELD x)   (FIELD y)   = f x y
```

```
gEq{|PAIR|}   fl fr (PAIR lx rx) (PAIR ly ry) = fl lx ly && fr rx ry
gEq{|EITHER|} fl _  (LEFT x)     (LEFT y)     = fl x y
gEq{|EITHER|} _  fr (RIGHT x)    (RIGHT y)    = fr x y
gEq{|EITHER|} _  _  _            _            = False

:: T = C1 Int ([Char], ?Bool) | C2
derive gEq [], T, (,), ?

Start = (gEq{|*|} C2 (C1 42 ([], ?Just True))
    , gEq{|*→*|} (<) [1,2,3] [2,3,4])
// (False, True)
```

Listing (Clean) A.4: Generic equality function

Metadata about the types is available using the **of** syntax that gives the function access to metadata records, as can be seen in listing A.5 showing a generic print function. This abundance of metadata allows for very complex generic functions that near the expression level of template metaprogramming (see chapter 3 and section 7.4).

```
generic gPrint a :: a [String] → [String]

gPrint{|Int|}          x         acc = [toString x:acc]
gPrint{|Bool|}         x         acc = [toString x:acc]
gPrint{|Real|}         x         acc = [toString x:acc]
gPrint{|Char|}         x         acc = [toString x:acc]
gPrint{|UNIT|}         x         acc = acc
gPrint{|PAIR|}   fl fr (PAIR l r) acc = fl l [" ":fr r acc]
gPrint{|EITHER|} fl _  (LEFT x)   acc = fl x acc
gPrint{|EITHER|} _  fr (RIGHT x)  acc = fr x acc

gPrint{|OBJECT|}       f     (OBJECT x) acc = f x acc
gPrint{|CONS of gcd|}  f     (CONS x)   acc
    = ["(", gcd.gcd_name, " ":f x [")":acc]]
gPrint{|RECORD of grd|} f    (RECORD x) acc
    = ["{", grd.grd_name, " / ":f x ["}":acc]]
gPrint{|FIELD of gfd|}  f    (FIELD x)  acc
    = [pre, gfd.gfd_name, "=":f x acc]
where pre = if (gfd.gfd_index == 0) "" ", "

:: T = {f1 :: Int, f2 :: (Real, [?Int])}
derive gPrint (,), [], ?, T

Start = gPrint{|*|} {f1=42, f2=(3.14, [?None])} []
// {T | f1=42 , f2=(_Tuple2 3.14 (_Cons (_!None ) (_Nil )))}
```

Listing (Clean) A.5: Generic print function

A.1.6 GADTs

GADTs are enriched data types that allow the type instantiation of the constructor to be explicitly defined (Cheney and Hinze, 2003; Hinze, 2003). While GADTs are not natively supported in Clean, they can be simulated using embedding-projection pairs or equivalence types (Cheney and Hinze, 2002, section 2.2). To illustrate this, listing A.6 shows a GADT in Haskell[3] that can be implemented as in listing A.7.

```haskell
data Expr a where
    Lit :: Show a ⇒ a → Expr a
    Add :: Num a  ⇒ Expr a → Expr a → Expr a
    Eq  :: Eq e   ⇒ Expr e → Expr e → Expr Bool

eval :: Expr a → a
eval (Lit e)   = e
eval (Add l r) = eval l + eval r
eval (Eq  l r) = eval l == eval r

print :: Expr a → String
print (Lit e)   = show e
print (Add l r) = print l ++ "+"  ++ print r
print (Eq  l r) = print l ++ "==" ++ print r
```
<div align="center">Listing (Haskell) A.6: Expression GADT.</div>

```clean
:: BM a b = { ab :: a → b, ba  :: b → a }
bm :: BM a a
bm = {ab=id, ba=id}

:: Expr a
    = E.e: Lit (BM a e) e                       & toString e
    | E.e: Add (BM a e) (Expr e) (Expr e)    & + e
    | E.e: Eq  (BM a Bool) (Expr e) (Expr e) & == e
lit e = Lit bm e
add l r = Add bm l r
eq l r = Eq bm l r

eval :: (Expr a) → a
eval (Lit bm e)   = bm.ba e
eval (Add bm l r) = bm.ba (eval l + eval r)
eval (Eq  bm l r) = bm.ba (eval l == eval r)

print :: (Expr a) → String
print (Lit _ e)   = toString e
print (Add _ l r) = print l +++ "+" +++ print r
print (Eq  _ l r) = print l +++ "==" +++ print r
```
<div align="center">Listing (Clean) A.7: Expression GADT using equivalence types.</div>

---

[3]Requires `GADTs` to be enabled.

A.2 Syntax

Table A.1: Syntactical differences between Clean and Haskell.

Clean	Haskell
Comments	
// single line	*−− single line*
/∗ multi line /∗ nested ∗/ ∗/	*{− multi line {− nested −} }*
Imports	
import Mod ⇒ **qualified** f1, :: t	**import qualified** Mod (f1, t)
	import Mod **hiding** (f1, t)
Basic types	
42 :: Int	42 :: Int
True :: Bool	True :: Bool
toInteger 42 :: Integer	42 :: Integer
38.0 :: Real	38.0 :: Float *−− or Double*
"Hello" +++ *"World"* :: String[4]	*"Hello"* ++ *"World"* :: String[5]
['Hello'] :: [Char]	*"Hello"* :: String
?t	Maybe t
(?None, ?Just e)	(Nothing, Just e)
Type definitions	
:: T a0 ⋯ :== t	**type** T a0 ⋯ = t
:: T a0 ⋯	**data** T a0 ⋯
= C1 f0 ⋯ fn | ⋯ | Cn f0 ⋯ fn	= C1 f0 ⋯ fn | ⋯ | Cn f0 ⋯ fn
:: T a0 ⋯	**data** T a0 ⋯
= { f0 :: t0, ⋯, fn :: tn }	= T { f0 :: t0, ⋯, fn :: tn }
:: T a0 ⋯ =: t	**newtype** T a0 ⋯ = t
:: T = E.t: Box t & C t	**data** T = **forall** t.C t ⇒ Box t[6]
Function types	
f0 :: a0 a1 ⋯ → t	f0 :: (c0 v0, c1 v1, c2 v2) ⇒
| c0 v0 & c1, c2 v1	a0 → a1 ⋯ → t
(+) **infixl** 6 :: Int Int → Int	**infixl** 6 +
	(+) :: Int → Int → Int
qid :: (A.a: a → a) → (Bool, Int)	qid :: (**forall** a: a → a) → (Bool, Int)[7]

---

[4]Strings are unboxed character arrays.

[5]Strings are lists of characters by default but may be overloaded as well if `OverloadedStrings` is enabled.

[6]Requires `ExistentialQuantification` to be enabled.

[7]Requires `RankNTypes` to be enabled.

Table A.1: Syntactical differences between Clean and Haskell. (continued)

Clean	Haskell
`qid id = (id True, id 42)`	`qid id = (id True, id 42)`

Type classes	
class f a :: t	**class** f a **where** f :: t
class C a | C0, $\cdots$, Cn a[8]	**class** (C0 a, $\cdots$, Cn, a) $\Rightarrow$ C a
class C s ~m **where** $\cdots$	**class** C s m | m $\rightarrow$ s **where** $\cdots$ [9]
instance C t | C0, $\cdots$, Cn a	**instance** (C0 a, $\cdots$, Cn a) $\Rightarrow$ C t
where $\cdots$	**where** $\cdots$

As pattern	
`x=:p`	`x@p`

Lists	
`[1,2,3]`	`[1,2,3]`
`[x:xs]`	`x:xs`
`[e \\ e <- xs	p e]`	`[e	e ← xs, p e]`
`[l \\ l <- xs, r <- ys]`	`[l	l ← xs, r ← ys]`
`[(l, r) \\ l <- xs & r <- ys]`	`[(l, r)	(l, r) ← zip xs ys]`
	or `[(l, r)	l ← xs	r ← ys]`[10]

Lambda expressions	
`\a0 a1` $\cdots \rightarrow$`e` or `\`$\cdots$`.e` or `\`$\cdots$`=e`	`\a0 a1` $\cdots \rightarrow$`e`

Case distinction	
if p e0 e1	**if** p **then** e0 **else** e1
case e **of** p0 $\rightarrow$ e0; $\cdots$	**case** e **of** p0 $\rightarrow$ e0; $\cdots$
or **case** e **of** p0 = e0; $\cdots$	
f p0 $\cdots$ pn	f p0 $\cdots$ pn
| c = t	| c = t
| **otherwise** = t or = t	| **otherwise** = t

---

[8]In contrast to the Haskell variant, this does not require an instance.

[9]Requires `MultiParamTypeClasses` to be enabled.

[10]Requires `ParallelListComp` to be enabled.

Table A.1: Syntactical differences between Clean and Haskell. (continued)

Clean	Haskell
Record expressions	
```:: R = { f :: t }```	```data R = R { f :: t }```
```r = { f = e }```	```r = R { f = e }```
```r.f```	```f r```
	```or r.f```[11]
```r!f```[12]	```(\v→(f v, v)) r```
```{r & f = e }```	```r { f = e }```
Record patterns	
```:: R0 = { f0 :: R1 Int }```	```data R0 = R0 { f0 :: R1 Int }```
```:: R1 t = { f1 :: t }```	```data R1 t = R1 { f1 :: t }```
```g { f0 } = e f0```	```g (R0 {f0=x}) = e x```
	```or g (R0 {f0}) = e f0```[13]
```g { f0 = {f1} } = e f1```	```g (R0 {f0=R1 {f1=x}}) = e x```
Arrays	
```:: A :== {t}```	```type A = Array Int t```
```a = {v0, ⋯, vn}```	```a = array (0, n+1)```
	```[(0, v0), ⋯, (n, vn)]```
```a = {e \\ p <-: a}```	```a = array (0, length a-1)```
	``[e	(i, a) ← zip [0..] a]```
```a.[i]```	```a!i```
```a![i]```[14]	```(\v→(v!i, v)) a```
```{ a & [i] = e}```	```a//[(i, e)]```
Dynamics	
``f :: a → Dynamic	TC a```	```f :: Typeable a ⇒ a → Dynamic```
```f e = dynamic e```	```f e = toDyn (e)```
```g :: Dynamic → t```	```g :: Dynamic → t```
```g (e :: t) = e0```	```g d = case fromDynamic d```
```g e = e1```	```Just e → e0```
	```Nothing → e1```

---

[11]Requires `OverloadedRecordDot` to be enabled. Requires GHC version 9.2.0 or higher
[12]This operator allows for field selection from unique records.
[13]Requires `RecordPuns` to be enabled.
[14]This operator allows for field selection from unique arrays.

Appendix B

Auxiliary mTask type classes

B.1   Peripherals

This section shows the peripherals not mentioned in chapter 4. All constructors use HOAS to create a type safe sensor object from a connection specification that can be used to interact with the sensor. The measurement tasks all yield unstable values containing the measured value. The auxiliary functions such as calibration yield stable values indicating the result. Tasks suffixed with the backtick (`'`) indicate variants for which the timing interval can be specified (see chapter 8).

B.1.1   Air quality sensor

The mTask language supports one type (*CCS811* connected via I^{2}C) of air quality sensors that measures total volatile organic compounds (TVOC) (ppm) and eCO$_2$ (%). Besides the constructor and tasks for the measurements there is also a calibration task that can be used to calibrate the sensor from temperature and humidity readings to increase the accuracy. The complete interface is shown in listing B.1.

```
:: AirQualitySensor // abstract

class AirQualitySensor v where
    airqualitysensor :: I2CAddr ((v AirQualitySensor) → Main (v a)) → Main (v a)
    tvoc` :: (TimingInterval v) (v AirQualitySensor) → MTask v Int
    tvoc :: (v AirQualitySensor) → MTask v Int
    co2` :: (TimingInterval v) (v AirQualitySensor) → MTask v Int
    co2 :: (v AirQualitySensor) → MTask v Int
    setEnvironmentalData :: (v AirQualitySensor) (v Real) (v Real) → MTask v ()
    setEnvFromDHT :: (v AirQualitySensor) (v DHT) → MTask v ()
```

Listing (Clean) B.1: Air quality sensor interface in mTask.

B.1.2 Gesture sensor

The mTask language supports one type (*PAJ7620* connected via I²C) of gesture
sensors. The *PAJ7620* contains an optical CMOS array that measures the reflection
of the on-board IR LED to detect up to several gestures. The complete interface
containing the constructor and the measurement task is shown in listing B.2.

```
:: GestureSensor // abstract
:: Gesture = GNone | GRight | GLeft | GUp | GDown | GForward | GBackward
           | GClockwise | GCountClockwise

class GestureSensor v where
    gestureSensor :: I2CAddr ((v GestureSensor) → Main (v a)) → Main (v a)
    gesture` :: (TimingInterval v) (v GestureSensor) → MTask v Gesture
    gesture :: (v GestureSensor) → MTask v Gesture
```

Listing (Clean) B.2: Gesture sensor interface in mTask.

B.1.3 Light intensity sensor

The mTask language supports one type (*BH1750* connected via I²C) of light
intensity sensors that measure the light intensity in lx. The complete interface
containing the constructor and the measurement task is shown in listing B.3.

```
:: LightSensor // abstract

class LightSensor v where
    lightsensor :: I2CAddr ((v LightSensor) → Main (v b)) → Main (v b)
    light` :: (TimingInterval v) (v LightSensor) → MTask v Real
    light :: (v LightSensor) → MTask v Real
```

Listing (Clean) B.3: Light intensity sensor interface in mTask.

B.1.4 Motion detection sensor

The mTask language supports motion sensing using a PIR sensor through a type
class that only contains macros. PIR sensors detect motion by the IR reflection
through a number of Fresnel lenses and communicates through a digital GPIO
pin. Therefore, a PIR is nothing more than a DPin according to mTask but for
uniformity, a type class is available (see listing B.4).

```
:: PIR :== DPin

class PIR v | step, expr, pinMode v & dio DPin v where
    PIR :: DPin ((v PIR) → Main (v b)) → Main (v b) | expr, step, pinMode v

    motion` :: (TimingInterval v) (v PIR) → MTask v Bool
    motion :: (v PIR) → MTask v Bool | dio DPin v
```

Listing (Clean) B.4: PIR sensor interface in mTask.

B.1.5 Sound detection sensor

The mTask language supports motion sensing using one type of sensor (*SEN-12642*) that outputs either a gate value through a digital GPIO pin, the envelope (amplitude) through an analog GPIO pin and an audio output. Only the sound level—i.e. the envelope—and the sound presence are available in mTask through a type class containing only macros. Therefore, a sound detector is nothing more than a tuple of a `DPin` for the gate value and an `APin` for the envelope (see listing B.5).

```clean
:: SoundDetector :== (DPin, APin)

class SoundDetector v | tupl, expr, pinMode v & dio DPin v where
    soundDetector :: DPin APin ((v SoundDetector) → Main (v b)) → Main (v b)

    soundPresence` :: (TimingInterval v) (v SoundDetector) → MTask v Bool
    soundPresence :: (v SoundDetector) → MTask v Bool | tupl v & dio DPin v

    soundLevel` :: (TimingInterval v) (v SoundDetector) → MTask v Bool
    soundLevel :: (v SoundDetector) → MTask v Bool | tupl, aio v
```
Listing (Clean) B.5: Sound detection sensor interface in mTask.

B.1.6 I²C buttons

The mTask language supports one type of I²C buttons (the I²C buttons from the WEMOS D1 mini OLED shield). The buttons from this shield provide more information than just the status (see `ButtonStatus`). The complete interface containing the constructor and the measurement tasks is shown in listing B.6.

```clean
:: I2CButton   // abstract
:: ButtonStatus = ButtonNone | ButtonPress | ButtonLong | ButtonDouble | ButtonHold

class i2cbutton v where
    i2cbutton :: I2CAddr ((v I2CButton) → Main (v b)) → Main (v b) | type b

    AButton` :: (TimingInterval v) (v I2CButton) → MTask v ButtonStatus
    AButton :: (v I2CButton) → MTask v ButtonStatus

    BButton` :: (TimingInterval v) (v I2CButton) → MTask v ButtonStatus
    BButton :: (v I2CButton) → MTask v ButtonStatus
```
Listing (Clean) B.6: I²C button interface in mTask.

B.1.7 LED matrix

The mTask language supports one type of LED matrix (the *8 × 8* LED matrix shield for the WEMOS D1 mini). Instead of containing a TOP-like interface, the Arduino interface is directly translated to mTask. As a result, every task immediately returns a stable value indicating the result. The complete interface containing the constructor and the interaction tasks is shown in listing B.7.

```clean
:: LEDMatrix // abstract
:: LEDMatrixInfo = { dataPin :: Pin, clockPin :: Pin }
```

```
class LEDMatrix v where
    ledmatrix :: LEDMatrixInfo ((v LEDMatrix) → Main (v b)) → Main (v b) | type b
    LMDot :: (v LEDMatrix) (v Int) (v Int) (v Bool) → MTask v ()
    LMIntensity :: (v LEDMatrix) (v Int) → MTask v ()
    LMClear :: (v LEDMatrix) → MTask v ()
    LMDisplay :: (v LEDMatrix) → MTask v ()
```

Listing (Clean) B.7: LED matrix interface in mTask.

B.1.8   Connection types

The connection between the iTask server and the mTask devices are communication method agnostic. As long as the `channelSync` type class is implemented, the communication method can be used. Listing B.8 shows the data types for the connections.

```
:: TCPSettings =
    { host        :: String
    , port        :: Int
    , pingTimeout :: ?Int
    }
:: MQTTSettings =
    { host     :: String
    , port     :: Int
    , mcuId    :: String
    , serverId :: String
    , auth     :: MQTTAuth
    }
:: TTYSettings =
    { devicePath :: String
    , baudrate   :: BaudRate
    , bytesize   :: ByteSize
    , parity     :: Parity
    , stop2bits  :: Bool
    , xonxoff    :: Bool
    , sleepTime  :: Int
    }
```

Listing (Clean) B.8: Data types for the different connections in mTask.

Appendix C

Bytecode instruction set

Tasks in mTask are compiled at run time to byte code. This byte code is evaluated using the interpreter. The result of this evaluation is a task tree. Subsequently, this task tree is rewritten until a stable value is observed. This appendix describes the semantics of the byte code instruction set of mTask (see chapter 7). The byte code instructions are of variable length and automatically encoded and decoded using generic programming (see section 7.4). Table C.1 shows the notational convention of the variables used in the table. Table C.2 shows the semantics of all major byte code instructions, shorthand instructions and auxiliary peripherals have been omitted for brevity but have the analogous semantics as their counterparts.

Table C.1: Notation convention for the byte code semantics.

variable	meaning	№bytes
fp	frame pointer	2
sp	stack pointer	2
pc	program counter	2
l	label	2
w_r	return width	1
w_a	argument width	1
i	SDS or sensor id	1
n	number	1
d	depth	1

Table C.2: Semantics for the bytecode instructions.

Instruction	Arguments	Semantics	sp	pc
push	$n\ b_0 \ldots b_n$	$st[sp + i] = s[i]$ **for all** $i \in \{0..n\}$	$sp + n$	$pc + 2 + n$
pop	n		$sp - n$	$pc + 2$
rot	$d\ n$	$rotate\,(d, n)$	sp	$pc + 3$
dup		$st[sp] = st[sp$ - $1]$	$sp + 1$	$pc + 1$
jumpF	l		$sp - 1$	$\begin{cases} pc + 1 & \textbf{if } st[sp$ - $1] \\ l & \textbf{otherwise} \end{cases}$
jump	l		$sp - 1$	l
jumpSR	$w_a\ l$	$st[sp$ - w_a - $1] = pc + 2$		l
tailCall	$w_{a_1}\ w_{a_2}\ l$	$rotate\,(w_{a_1} + 3 + w_{a_2}, w_{a_2})$	fp	jl
		$fp = fp$ - $w_{a_1} + w_{a_2}$		
		where w_{a_1} is the width of the current function and w_{a_2} the		
		width of the called function		
arg	n	$st[sp] = st[fp$ - 1 - $n]$	$sp + 1$	
return	$w_r\ w_a$	$st[fp$ - w_a - $3 + i] = st[fp + 1]$	$st[fp$ - w_a - $3 + w_r]$	$st[fp$ - w_a - $1]$
		for all $i \in \{0..w_r\}$		
		$fp = st[fp$ - w_a - $2]$		
pushPtrs		$st[sp] = sp$	$sp + 3$	$pc + 1$
		$st[sp + 1] = fp$		
		$st[sp + 2] = 0$		
unOp		$st[sp$ - $1] = \diamond st[sp$ - $1]$ **for all** $\diamond \in \{\neg\}$	sp	$pc + 1$
binOp		$st[sp$ - $2] = st[sp$ - $2] \oplus st[sp$ - $1]$	$sp - 1$	$pc + 1$
		for all $\oplus \in \{+, $-$, *, /, \wedge, \vee, \equiv, \not\equiv, \leq, \geq, <, >\}$		
		similar for Real and Long variants		
$cast_{f\text{-}t}$		$st[sp$ - $1] = cast_{f--t}(st[sp$ - $1])$	sp	$pc + 1$
		for all $f, t \in \{Int, Real, Long\}$		

Table C.2: Semantics for the bytecode instructions. (continued)

Instruction	Arguments	Semantics	sp	pc
mkTask	Stable$_n$	$st[sp - n - 1] = node(stable,$ $st[sp - 1], \ldots, st[sp - n - 1])$	$sp - n + 1$	$pc + 2$
mkTask	Unstable$_n$	$st[sp - n - 1] = node(unstable,$ $st[sp - 1], \ldots, st[sp - n - 1])$	$sp - n + 1$	$pc + 2$
mkTask	ReadD	$st[sp - 1] = node(readd, st[sp - 1])$	sp	$pc + 2$
mkTask	ReadA	$st[sp - 1] = node(reada, st[sp - 1])$	sp	$pc + 2$
mkTask	WriteD	$st[sp - 2] = node(writed, st[sp - 1], st[sp - 2])$	$sp - 1$	$pc + 2$
mkTask	WriteA	$st[sp - 2] = node(writea, st[sp - 1], st[sp - 2])$	$sp - 1$	$pc + 2$
mkTask	WriteD	$st[sp - 2] = node(writed, st[sp - 1], st[sp - 2])$	$sp - 1$	$pc + 2$
mkTask	PinMode	$st[sp - 2] = node(pinmode, st[sp - 1], st[sp - 2])$	$sp - 1$	$pc + 2$
mkTask	Repeat	$st[sp] = node(repeat, st[sp - 1])$	sp	$pc + 2$
mkTask	Delay	$st[sp] = node(delay, st[sp - 1])$	sp	$pc + 2$
mkTask	And	$st[sp - 1] = node(and, st[sp - 1], st[sp - 2])$	$sp - 1$	$pc + 2$
mkTask	Or	$st[sp - 1] = node(and, st[sp - 1], st[sp - 2])$	$sp - 1$	$pc + 2$
mkTask	Step $f\ w_a$	$st[sp] = node(step, st[sp - 1], f, w)$	sp	$pc + 5$
mkTask	SdsGet i	$st[sp + 1] = node(sdsget, i)$	$sp + 1$	$pc + 3$
mkTask	SdsSet i	$st[sp - 1] = node(sdsset, st[sp - 1], i)$	sp	$pc + 3$
mkTask	SdsUpd $i\ l$	$st[sp + 1] = node(sdsset, i, l)$	$sp + 1$	$pc + 5$
mkTask	Interrupt	$st[sp - 2] = node(interrupt, st[sp - 1], st[sp - 2])$	$sp - 1$	$pc + 2$
mkTask	RateLimit	$st[sp - 1] = node(ratelimit, st[sp - 1])$	sp	$pc + 2$
mkTask	TuneRate	$st[sp - 1] = node(tunerate, st[sp - 1], st[sp - 2])$	$sp - 1$	$pc + 2$
mkTask	DHTTemp i	$st[sp + 1] = node(dhttemp, i)$	$sp + 1$	$pc + 3$
mkTask	DHTHumid i	$st[sp + 1] = node(dhthumid, i)$	$sp + 1$	$pc + 3$

Bibliography

van der Aalst, W., A. ter Hofstede, B. Kiepuszewski and A. Barros (2003). 'Workflow Patterns'. In: *Distributed and Parallel Databases* 14.1, pp. 5–51. ISSN: 1573-7578. DOI: 10.1023/A:1022883727209 (cit. on p. 137).

Abadi, M., L. Cardelli, B. Pierce and G. Plotkin (1991). 'Dynamic Typing in a Statically Typed Language'. In: *ACM Trans. Program. Lang. Syst.* 13.2. Place: New York, NY, USA Publisher: ACM, pp. 237–268. ISSN: 0164-0925. DOI: 10.1145/103135.103138 (cit. on p. 28).

Achten, P. (2007). *Clean for Haskell98 Programmers*. URL: https://www.mbsd.cs.ru.nl/publications/papers/2007/achp2007-CleanHaskellQuickGuide.pdf (cit. on pp. 154, 187).

Achten, P., J. van Groningen and R. Plasmeijer (1993). 'High Level Specification of I/O in Functional Languages'. In: *Functional Programming, Glasgow 1992*. Ed. by J. Launchbury and P. Sansom. London: Springer London, pp. 1–17. ISBN: 978-1-4471-3215-8 (cit. on p. 188).

Achten, P. and R. Plasmeijer (1995). 'The ins and outs of Clean I/O'. In: *Journal of Functional Programming* 5.1. Publisher: Cambridge University Press, pp. 81–110. DOI: 10.1017/S0956796800001258 (cit. on p. 188).

Adams, M. D. and T. M. DuBuisson (2012). 'Template Your Boilerplate: Using Template Haskell for Efficient Generic Programming'. In: *Proceedings of the 2012 Haskell Symposium*. Haskell '12. event-place: Copenhagen, Denmark. New York, NY, USA: ACM, pp. 13–24. ISBN: 978-1-4503-1574-6. DOI: 10.1145/2364506.2364509 (cit. on p. 59).

Alhirabi, N., O. Rana and C. Perera (2021). 'Security and Privacy Requirements for the Internet of Things: A Survey'. In: *ACM Trans. Internet Things* 2.1. Place: New York, NY, USA Publisher: ACM. ISSN: 2691-1914. DOI: 10.1145/3437537 (cit. on pp. 135, 152).

Alimarine, A. (2005). 'Generic Functional Programming'. PhD thesis. Nijmegen: Radboud University. 198 pp. (cit. on pp. 115, 189).

Alimarine, A. and R. Plasmeijer (2002). 'A Generic Programming Extension for Clean'. In: *Implementation of Functional Languages*. Ed. by T. Arts and M. Mohnen. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 168–185. ISBN: 978-3-540-46028-2 (cit. on p. 154).

Alpernas, K., Y. M. Y. Feldman and H. Peleg (2020). 'The Wonderful Wizard of LoC: Paying Attention to the Man behind the Curtain of Lines-of-Code Metrics'. In: *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2020. Virtual, USA: ACM, pp. 146–156. ISBN: 9781450381789. DOI: 10.1145/3426428.3426921 (cit. on pp. 145, 166).

Alphonsa, M. (2021). 'A Review on IOT Technology Stack, Architecture and Its Cloud Applications in Recent Trends'. In: *ICCCE 2020*. Ed. by A. Kumar and S. Mozar. Singapore: Springer Singapore, pp. 703–711. ISBN: 978-981-15-7961-5 (cit. on p. 144).

Amazonas Cabral de Andrade, M. (2018). 'Developing Real Life, Task Oriented Applications for the Internet of Things'. Master's Thesis. Nijmegen: Radboud University. 60 pp. (cit. on p. 139).

St-Amour, V. and M. Feeley (2009). 'PICOBIT: a compact scheme system for microcontrollers'. In: *International Symposium on Implementation and Application of Functional Languages*. Springer, pp. 1–17 (cit. on pp. 133, 176).

Antonakakis, M., T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas and Y. Zhou (2017). 'Understanding the Mirai Botnet'. In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, pp. 1093–1110. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis (cit. on p. 153).

Antonova, E. (2022). 'mTask Semantics and its Comparison to TopHat'. Bachelor's Thesis. Nijmegen: Radboud University. 60 pp. (cit. on pp. 137, 140).

Ashton, K. (1999). 'Internet of Things'. Presentation. Presentation. Proctor & Gamble. London, UK (cit. on p. 3).

Ashton, K. (2009). 'That 'Internet of Things' Thing'. In: *RFID journal* 22.7. Publisher: Hauppauge, New York, pp. 97–114 (cit. on p. 3).

Atkey, R., S. Lindley and J. Yallop (2009). 'Unembedding Domain-Specific Languages'. In: *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*. Haskell '09. event-place: Edinburgh, Scotland. New York, NY, USA: ACM, pp. 37–48. ISBN: 978-1-60558-508-6. DOI: 10.1145/1596638.1596644 (cit. on p. 58).

Baaij, C. P. R. (2015). 'Digital circuit in CλaSH: functional specifications and type-directed synthesis'. ISBN: 978-90-365-3803-9. PhD thesis. Netherlands: University of Twente. DOI: 10.3990/1.9789036538039 (cit. on pp. 59, 136).

Baars, A. I. and S. D. Swierstra (2002). 'Typing Dynamic Typing'. In: *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*. ICFP '02. event-place: Pittsburgh, PA, USA. New York, NY, USA: ACM, pp. 157–166. ISBN: 1-58113-487-8. DOI: 10.1145/581478.581494 (cit. on p. 28).

Baccelli, E., J. Doerr, O. Jallouli, S. Kikuchi, A. Morgenstern, F. A. Padilla, K. Schleiser and I. Thomas (2018). 'Reprogramming Low-end IoT Devices from the Cloud'. In: *2018 3rd Cloudification of the Internet of Things (CIoT)*. IEEE, pp. 1–6 (cit. on pp. 132, 152).

Backus, J., J. H. Williams and E. L. Wimmers (1990). 'An Introduction to the Programming Language FL'. In: *Research Topics in Functional Programming*. USA: Addison-Wesley Longman Publishing Co., Inc., pp. 219–247. ISBN: 0-201-17236-4 (cit. on p. 188).

Balat, V. (2006). 'Ocsigen: Typing web interaction with objective caml'. In: *Proceedings of the 2006 Workshop on ML*, pp. 84–94 (cit. on p. 149).

Barendregt, H., M. van Eekelen, J. Glauert, J. Kennaway, M. Plasmeijer and M. Sleep (1987). 'Towards an intermediate language for graph rewriting'. In: *PARLE, Parallel Architectures and Languages Europe*. Vol. 1. Springer Verlag, pp. 159–174 (cit. on p. 187).

Barendsen, E. and S. Smetsers (1996). 'Uniqueness typing for functional languages with graph rewriting semantics'. In: *Mathematical structures in computer science* 6.6, pp. 579–612 (cit. on pp. 154, 188).

Barišic, A., V. Amaral, M. Goulão and B. Barroca (2014). 'Evaluating the Usability of Domain-Specific Languages'. In: *Software Design and Development: Concepts, Methodologies, Tools, and Applications*. Ed. by I. R. Management Association. Hershey, PA, USA: IGI Global, pp. 2120–2141. ISBN: 978-1-4666-4301-7. DOI: 10.4018/978-1-4666-4301-7.ch098 (cit. on p. 137).

Bawden, A. (1999). 'Quasiquotation in Lisp'. In: *O. Danvy, Ed., University of Aarhus, Dept. of Computer Science*. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation. Vol. NS-99-1. BRICS Notes Series. Aarhus, Denmark: BRICS, pp. 88–99. DOI: 10.1.1.22.1290 (cit. on p. 48).

Belle, A. B., G. El-Boussaidi, C. Desrosiers and H. Mili (2013). 'The layered architecture revisited: Is it an optimization problem?' In: *Proceedings of the Twenty-Fifth International Conference on Software Engineering & Knowledge E.* Vol. 1. Boston, MA, USA: KSI Research Inc, pp. 344–349. ISBN: 978-1-5108-4159-8 (cit. on p. 148).

Belwal, C., A. M. K. Cheng, J. Ras and Y. Wen (2013). 'Variable Voltage Scheduling with the Priority-Based Functional Reactive Programming Language'. In: *Proceedings of the 2013 Research in Adaptive and Convergent Systems*. RACS '13. event-place: Montreal, Quebec, Canada. New York, NY, USA: ACM, pp. 440–445. ISBN: 978-1-4503-2348-2. DOI: 10.1145/2513228.2513271 (cit. on p. 138).

Bjornson, J., A. Tayanovskyy and A. Granicz (2010). 'Composing reactive GUIs in F# using WebSharper'. In: *Symposium on Implementation and Application of Functional Languages*. Springer, pp. 203–216 (cit. on p. 149).

Blanchette, H., N. Vazou and L. Lampropoulos (2022). 'Liquid Proof Macros'. In: *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*. Haskell 2022. event-place: Ljubljana, Slovenia. New York, NY, USA: ACM, pp. 27–38. ISBN: 978-1-4503-9438-3. DOI: 10.1145/3546189.3549921 (cit. on p. 59).

de Boer, M. (2020). 'Secure Communication Channels for the mTask System.' Bachelor's Thesis. Nijmegen: Radboud University. 39 pp. (cit. on pp. 135, 140, 153).

Bolingbroke, M. (2011). *Constraint Kinds for GHC*. :: (Bloggable a) => a -> IO (). URL: http://blog.omega-prime.co.uk/2011/09/10/constraint-kinds-for-ghc/ (visited on 09/06/2021) (cit. on p. 25).

Boulton, R., A. Gordon, M. Gordon, J. Harrison, J. Herbert and J. V. Tassel (1992). 'Experience with embedding hardware description languages in HOL'. In: *IFIP TC10/WG*. Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience. Ed. by V. Stavridou, T. F. Melham and R. T. Boute. Vol. 10. event-place: Nijmegen, NL. North-Holland: Elsevier, pp. 129–156. ISBN: 0-444-89686-4 (cit. on p. 19).

Brus, T. H., M. C. J. D. van Eekelen, M. O. van Leer and M. J. Plasmeijer (1987). 'Clean — A language for functional graph rewriting'. In: *Functional Programming Languages and Computer Architecture*. Ed. by G. Kahn. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 364–384. ISBN: 978-3-540-47879-9 (cit. on p. 187).

Carette, J., O. Kiselyov and C.-C. Shan (2009). 'Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages'. In: *Journal of Functional Programming* 19.5. Publisher: Cambridge University Press, pp. 509–543. DOI: 10.1017/S0956796809007205 (cit. on pp. 20, 33, 43).

Cass, S. (2020). 'The top programming languages: Our latest rankings put Python on top-again-[Careers]'. In: *IEEE Spectrum* 57.8, pp. 22–22 (cit. on p. 175).

Cheney, J. and R. Hinze (2002). 'A lightweight implementation of generics and dynamics'. In: *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. event-place: Pittsburgh Pennsylvania, USA. ACM, pp. 90–104. DOI: 10.1145/581690.581698. (Visited on 15/05/2017) (cit. on pp. 30, 191).

Cheney, J. and R. Hinze (2003). *First-class phantom types*. TR2003-1901. Cornell University. URL: https://ecommons.cornell.edu/handle/1813/5614 (visited on 15/05/2017) (cit. on pp. 30, 191).

Chlipala, A. (2008). 'Parametric Higher-Order Abstract Syntax for Mechanized Semantics'. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. ICFP '08. event-place: Victoria, BC, Canada. New York, NY, USA: ACM, pp. 143–156. ISBN: 978-1-59593-919-7. DOI: 10.1145/1411204.1411226 (cit. on pp. 44, 78).

Chong, S., J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng and X. Zheng (2007). 'Secure web applications via automatic partitioning'. In: *ACM SIGOPS Operating Systems Review* 41.6, pp. 31–44 (cit. on p. 153).

CircuitPython Team (2022). *CircuitPython*. [Online; accessed 2-March-2022]. URL: https://circuitpython.org/ (cit. on p. 168).

Clifton-Everest, R., T. L. McDonell, M. M. T. Chakravarty and G. Keller (2014). 'Embedding Foreign Code'. In: *Practical Aspects of Declarative Languages*. Ed. by M. Flatt and H.-F. Guo. Cham: Springer International Publishing, pp. 136–151. ISBN: 978-3-319-04132-2 (cit. on p. 60).

Cooper, E., S. Lindley, P. Wadler and J. Yallop (2007). 'Links: Web Programming Without Tiers'. In: *Formal Methods for Components and Objects*. Ed. by F. S. de Boer, M. M. Bonsangue, S. Graf and W.-P. de Roever. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 266–296. ISBN: 978-3-540-74792-5 (cit. on pp. 12, 144, 149, 151, 174, 180).

Crooijmans, S. (2021). 'Reducing the Power Consumption of IoT Devices in Task-Oriented Programming'. Master's Thesis. Nijmegen: Radboud University. 78 pp. (cit. on pp. 140, 178).

Crooijmans, S., M. Lubbers and P. Koopman (2022). 'Reducing the Power Consumption of IoT with Task-Oriented Programming'. In: *Trends in Functional Programming*. Ed. by W. Swierstra and N. Wu. Cham: Springer International Publishing, pp. 80–99. ISBN: 978-3-031-21314-4. DOI: 10.1007/978-3-031-21314-4_5 (cit. on pp. 14, 130, 140).

Czarnecki, K., J. T. O'Donnell, J. Striegnitz and W. Taha (2004). 'DSL Implementation in MetaOCaml, Template Haskell, and C++'. In: *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*. Ed. by C. Lengauer, D. Batory, C. Consel and M. Odersky. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 51–72. ISBN: 978-3-540-25935-0. DOI: 10.1007/978-3-540-25935-0_4 (cit. on p. 58).

Decker, B. (2015). 'Tierless Programming for the Internet of Things'. Master's Thesis. United States: Brown University. 18 pp. DOI: 10.2172/1169934. URL: https://www.osti.gov/biblio/1169934 (cit. on p. 132).

Domoszlai, L., B. Lijnse and R. Plasmeijer (2014). 'Parametric Lenses: Change Notification for Bidirectional Lenses'. In: *Proceedings of the 26th International Symposium on Implementation and Application of Functional Languages*. IFL '14. Boston, MA, USA: ACM. ISBN: 9781450332842. DOI: 10.1145/2746325.2746333 (cit. on p. 154).

Dubé, D. (2000). 'BIT: A very compact Scheme system for embedded applications'. In: *Proceedings of the Fourth Workshop on Scheme and Functional Programming* (cit. on pp. 133, 176).

Duregård, J. and P. Jansson (2011). 'Embedded Parser Generators'. In: *Proceedings of the 4th ACM Symposium on Haskell*. Haskell '11. event-place: Tokyo, Japan. New York, NY, USA: ACM, pp. 107–117. ISBN: 978-1-4503-0860-1. DOI: 10.1145/2034675.2034689 (cit. on p. 59).

Egi, S., A. Kawata, M. Kori and H. Ogawa (2022). 'Embedding Non-linear Pattern Matching with Backtracking for Non-free Data Types into Haskell'. In: *New Generation Computing* 40.2, pp. 481–506. ISSN: 1882-7055. DOI: 10.1007/s00354-022-00177-z (cit. on p. 60).

Eisenberg, R. A. and J. Stolarek (2014). 'Promoting Functions to Type Families in Haskell'. In: *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*. Haskell '14. event-place: Gothenburg, Sweden. New York, NY, USA: ACM, pp. 95–106. ISBN: 978-1-4503-3041-1. DOI: 10.1145/2633357.2633361 (cit. on p. 60).

Ekblad, A. and K. Claessen (2014). 'A Seamless, Client-Centric Programming Model for Type Safe Web Applications'. In: *SIGPLAN Not.* 49.12, pp. 79–89. ISSN: 0362-1340. DOI: 10.1145/2775050.2633367 (cit. on pp. 149–151).

Elliott, C., S. Finne and O. de Moor (2003). 'Compiling embedded languages'. In: *Journal of Functional Programming* 13.3. Publisher: Cambridge University Press, pp. 455–481. DOI: 10.1017/S0956796802004574 (cit. on pp. 6, 42).

Elliott, C. and P. Hudak (1997). 'Functional reactive animation'. In: *ACM SIGPLAN Notices*. Vol. 32. ACM, pp. 263–273 (cit. on p. 134).

Elliott, T., L. Pike, S. Winwood, P. Hickey, J. Bielman, J. Sharp, E. Seidel and J. Launchbury (2015). 'Guilt free ivory'. In: *ACM SIGPLAN Notices*. Vol. 50. ACM, pp. 189–200 (cit. on pp. 133, 150).

Epstein, J., A. P. Black and S. Peyton Jones (2011). 'Towards Haskell in the Cloud'. In: *Proceedings of the 4th ACM Symposium on Haskell*. Haskell '11. Tokyo, Japan: ACM, pp. 118–129. ISBN: 9781450308601. DOI: 10.1145/2034675.2034690 (cit. on p. 173).

Evans, D. (2011). *The Internet of Things: How the Next Evolution of the Internet Is Changing Everything*. URL: https://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf (cit. on p. 3).

Feeley, M. and D. Dubé (2003). 'PICBIT: A Scheme system for the PIC microcontroller'. In: *Proceedings of the Fourth Workshop on Scheme and Functional Programming*. Citeseer, pp. 7–15 (cit. on pp. 133, 176).

Feijs, L. (2013). 'Multi-tasking and Arduino : why and how?' In: *8th International Conference on Design and Semantics of Form and Movement (DeSForM 2013)*. 8th International Conference on Design and Semantics of Form and Movement (DeSForM 2013). Ed. by L. L. Chen, T. Djajadiningrat, L. M. G. Feijs, S. Fraser, J. Hu, S. Kyffin and D. Steffen. Wuxi, China, pp. 119–127. ISBN: 978-90-386-3462-3 (cit. on pp. 69, 133).

Folmer, H. H., R. d. Groote and M. J. G. Bekooij (2022). 'High-Level Synthesis of Digital Circuits from Template Haskell and SDF-AP'. In: *Embedded Computer Systems: Architectures, Modeling, and Simulation*. Ed. by A. Orailoglu, M. Reichenbach and M. Jung. Cham: Springer International Publishing, pp. 3–27. ISBN: 978-3-031-15074-6 (cit. on p. 59).

Fowler, M. (2010). *Domain Specific Languages*. 1st. Addison-Wesley Professional. ISBN: 0-321-71294-3 (cit. on p. 5).

Furr, M. and J. S. Foster (2005). 'Checking Type Safety of Foreign Function Calls'. In: *SIGPLAN Not.* 40.6, pp. 62–72. ISSN: 0362-1340. DOI: 10.1145/1064978.1065019 (cit. on p. 173).

van Gemert, D. (2022). 'Task Oriented Programming in LUA'. Bachelor's Thesis. Nijmegen: Radboud University. 63 pp. (cit. on p. 12).

GHC Team (2021a). *Data.Dynamic*. URL: https://hackage.haskell.org/package/base-4.14.1.0/docs/Data-Dynamic.html (visited on 24/02/2021) (cit. on p. 28).

GHC Team (2021b). *GHC User's Guide Documentation*. URL: https://downloads.haskell.org/~ghc/latest/docs/users_guide.pdf (visited on 24/02/2021) (cit. on pp. 20, 24, 31, 46, 189).

Gibbons, J. and N. Wu (2014). 'Folding Domain-Specific Languages: Deep and Shallow Embeddings (Functional Pearl)'. In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. event-place: Gothenburg, Sweden. New York, NY, USA: ACM, pp. 339–347. ISBN: 978-1-4503-2873-9. DOI: 10.1145/2628136.2628138 (cit. on p. 22).

Gill, A. (2009). 'A Haskell Hosted DSL for Writing Transformation Systems'. In: *Domain-Specific Languages*. Ed. by W. M. Taha. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 285–309. ISBN: 978-3-642-03034-5 (cit. on p. 60).

Grebe, M. and A. Gill (2016). 'Haskino: A remote monad for programming the arduino'. In: *International Symposium on Practical Aspects of Declarative Languages*. Springer, pp. 153–168 (cit. on p. 132).

Grebe, M. and A. Gill (2019). 'Threading the Arduino with Haskell'. In: *Trends in Functional Programming*. Ed. by D. Van Horn and J. Hughes. Cham: Springer International Publishing, pp. 135–154. ISBN: 978-3-030-14805-8 (cit. on p. 132).

van Groningen, J., T. van Noort, P. Achten, P. Koopman and R. Plasmeijer (2010). 'Exchanging sources between Clean and Haskell: A double-edged front end for the Clean compiler'. In: *ACM Sigplan Notices* 45.11, pp. 49–60 (cit. on p. 187).

Guinard, D. and V. Trifa (2016). *Building the Web of Things: With Examples in Node.Js and Raspberry Pi*. 1st. USA: Manning Publications Co. ISBN: 1-61729-268-0 (cit. on p. 149).

Gupta, M. (2012). *Akka essentials*. Livery Place, 35 Livery Street, Birmingham B3 2PB, UK: Packt Publishing Ltd (cit. on p. 173).

Haenisch, T. (2016). 'A case study on using functional programming for internet of things applications'. In: *Athens Journal of Technology & Engineering* 3.1 (cit. on p. 133).

Hall, C., K. Hammond, W. Partain, S. L. Peyton Jones and P. Wadler (1993). 'The Glasgow Haskell compiler: a retrospective'. In: *Functional Programming, Glasgow 1992*. Springer, pp. 62–71 (cit. on p. 150).

Hammond, K., J. Berthold and R. Loogen (2003). 'Automatic Skeletons in Template Haskell'. In: *Parallel Processing Letters* 13.3, pp. 413–424. DOI: 10.1142/S0129626403001380 (cit. on p. 59).

Harth, N., C. Anagnostopoulos and D. Pezaros (2018). 'Predictive intelligence to the edge: impact on edge analytics'. In: *Evolving Systems* 9.2, pp. 95–118 (cit. on p. 146).

HaskellWiki contributors (2020). *Introduction to IO — HaskellWiki*. [Online; accessed 19-January-2021]. URL: https://wiki.haskell.org/index.php?title=Introduction_to_IO&oldid=63493 (cit. on p. 154).

van der Heijden, M., B. Lijnse, P. J. Lucas, Y. F. Heijdra and T. R. Schermer (2011). 'Managing COPD exacerbations with telemedicine'. In: *Conference on Artificial Intelligence in Medicine in Europe*. Springer, pp. 169–178 (cit. on p. 135).

Helbling, C. and S. Z. Guyer (2016). 'Juniper: a functional reactive programming language for the Arduino'. In: *Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design*. ACM, pp. 8–16 (cit. on pp. 135, 150).

Hentschel, K., D. Jacob, J. Singer and M. Chalmers (2016). 'Supersensors: Raspberry Pi Devices for Smart Campus Infrastructure'. In: *2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud)*. 2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud). Vienna, Austria: IEEE, pp. 58–62. ISBN: 978-1-5090-4052-0. DOI: 10.1109/FiCloud.2016.16. (Visited on 04/09/2019) (cit. on pp. 15, 145, 146).

Herwig, S., K. Harvey, G. Hughey, R. Roberts and D. Levin (2019). 'Measurement and Analysis of Hajime, a Peer-to-peer IoT Botnet'. In: *Network and Distributed Systems Security (NDSS) Symposium 2019*. San Diego, CA, USA, p. 15. ISBN: 1-891562-55-X. DOI: 10.14722/ndss.2019.23488 (cit. on p. 153).

Hess, J. (2020). *arduino-copilot: Arduino programming in haskell using the Copilot stream DSL*. URL: https://hackage.haskell.org/package/arduino-copilot (visited on 14/02/2020) (cit. on pp. 133, 135, 150).

Hester, J. and J. Sorber (2019). 'Batteries Not Included'. In: *XRDS* 26.1. Place: New York, NY, USA Publisher: ACM, pp. 23–27. ISSN: 1528-4972. DOI: 10.1145/3351474 (cit. on p. 136).

Hinze, R. (2000). 'A New Approach to Generic Functional Programming'. In: *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '00. Boston, MA, USA: ACM, pp. 119–132. ISBN: 1581131259. DOI: 10.1145/325694.325709 (cit. on p. 154).

Hinze, R. (2003). 'Fun With Phantom Types'. In: *The Fun of Programming*. Ed. by J. Gibbons and O. de Moor. Cornerstones of Computing. Palgrave: Bloomsbury Publishing, pp. 245–262. ISBN: 978-0-333-99285-2 (cit. on pp. 30, 61, 191).

Hinze, R. and J. Jeuring (2003). 'Generic Haskell: Practice and Theory'. In: *Generic Programming: Advanced Lectures*. Ed. by R. Backhouse and J. Gibbons. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–56. ISBN: 978-3-540-45191-4. DOI: 10.1007/978-3-540-45191-4_1 (cit. on p. 189).

Hinze, R. and S. Peyton Jones (2001). 'Derivable Type Classes'. In: *Electronic Notes in Theoretical Computer Science* 41.1, pp. 5–35. ISSN: 1571-0661. DOI: https://doi.org/10.1016/S1571-0661(05) 80542-0 (cit. on p. 115).

Hudak, P. (1998). 'Modular domain specific languages and tools'. In: *Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203)*, pp. 134–142. DOI: 10.1109/ICSR.1998. 685738 (cit. on p. 6).

Hughes, J. (1989). 'Why functional programming matters'. In: *The computer journal* 32.2, pp. 98–107 (cit. on p. 172).

Ireland, C., D. Bowers, M. Newton and K. Waugh (2009). 'A Classification of Object-Relational Impedance Mismatch'. In: *First International Conference on Advances in Databases, Knowledge, and Data Applications*. First International Conference on Advances in Databases, Knowledge, and Data Applications. Cancun, Mexico: IEEE, pp. 36–43. ISBN: 978-0-7695-3550-0. DOI: 10.1109/ DBKDA.2009.11 (cit. on pp. 4, 148, 170, 173).

Jansen, J. M., B. Lijnse and R. Plasmeijer (2010). 'Towards dynamic workflows for crisis management'. In: *7th Proceedings of the International Conference on Information Systems for Crisis Response and Management, Seattle, WA, USA, May, 2010*. 7th International ISCRAM Conference on Information Systems for Crisis Response and Management. Ed. by S. French, n. Tomaszewski and C. Zobel. Seattle, USA: Information Systems for Crisis Response and Management, ISCRAM. ISBN: 978-972-49-2247-8 (cit. on p. 135).

Jeuring, J. and P. Jansson (1996). 'Polytypic programming'. In: *Advanced Functional Programming*. Ed. by J. Launchbury, E. Meijer and T. Sheard. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 68–114. ISBN: 978-3-540-70639-7 (cit. on p. 189).

Johnson-Davies, D. (2020). *Lisp for microcontrollers*. Lisp for microcontrollers. URL: https://ulisp.com (visited on 14/02/2020) (cit. on pp. 133, 176).

Kariotis, P. S., A. M. Procter and W. L. Harrison (2008). 'Making Monads First-Class with Template Haskell'. In: *Proceedings of the First ACM SIGPLAN Symposium on Haskell*. Haskell '08. event-place: Victoria, BC, Canada. New York, NY, USA: ACM, pp. 99–110. ISBN: 978-1-60558-064-7. DOI: 10.1145/1411286.1411300 (cit. on p. 60).

Kiselyov, O. (2011). 'Implementing Explicit and Finding Implicit Sharing in Embedded DSLs'. In: *Electronic Proceedings in Theoretical Computer Science* 66. Publisher: Open Publishing Association, pp. 210–225. DOI: 10.4204/eptcs.66.11 (cit. on p. 37).

Kiselyov, O. (2012). 'Typed Tagless Final Interpreters'. In: *Generic and Indexed Programming: International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*. Ed. by J. Gibbons. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 130–174. 978-3-642-32202-0. DOI: 10.1007/978-3-642-32202-0_3 (cit. on pp. 20, 42).

Kochhar, P. S., D. Wijedasa and D. Lo (2016). 'A Large Scale Study of Multiple Programming Languages and Code Quality'. In: *23rd International Conference on Software Analysis, Evolution, and Reengineering*. Osaka, Japan: IEEE, pp. 563–573. DOI: 10.1109/SANER.2016.112 (cit. on p. 149).

Kodali, R. K. and K. S. Mahesh (2016). 'Low cost ambient monitoring using ESP8266'. In: *2016 2nd International Conference on Contemporary Computing and Informatics (IC3I)*. IEEE. Greater Noida, India: IEEE, pp. 779–782 (cit. on p. 163).

Kohlbecker, E., D. P. Friedman, M. Felleisen and B. Duba (1986). 'Hygienic Macro Expansion'. In: *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*. LFP '86. event-place: Cambridge, Massachusetts, USA. New York, NY, USA: ACM, pp. 151–161. ISBN: 0-89791-200-4. DOI: 10.1145/319838.319859 (cit. on p. 47).

Koopman, P., M. Lubbers and R. Plasmeijer (2018). 'A Task-Based DSL for Microcomputers'. In: *Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL2018*. the Real World Domain Specific Languages Workshop 2018. Vienna, Austria: ACM Press, pp. 1–11. ISBN: 978-1-4503-6355-6. DOI: 10.1145/3183895.3183902. (Visited on 14/01/2019) (cit. on pp. 10, 13, 139, 176).

Koopman, P., M. Lubbers and R. Plasmeijer (2023). 'Simulation of a Task-Based Embedded Domain Specific Language for the Internet of Things'. In: *Composability, Comprehensibility and Correctness of Working Software, 7th Winter School, Košice, Slovakia, January 22–26, 2018, Revised Selected*

Papers. Lecture Notes in Computer Science 11916. in-press. Cham: Springer, p. 51 (cit. on pp. 14, 139).

Koopman, P., S. Michels and R. Plasmeijer (2021). 'Dynamic Editors for Well-Typed Expressions'. In: *Trends in Functional Programming*. Ed. by V. Zsók and J. Hughes. Cham: Springer International Publishing, pp. 44–66. ISBN: 978-3-030-83978-9 (cit. on p. 137).

Koopman, P. and R. Plasmeijer (2019). 'Type-Safe Functions and Tasks in a Shallow Embedded DSL for Microprocessors'. In: *Central European Functional Programming School: 6th Summer School, CEFP 2015, Budapest, Hungary, July 6–10, 2015, Revised Selected Papers*. Ed. by V. Zsók, Z. Porkoláb and Z. Horváth. Cham: Springer International Publishing, pp. 283–340. ISBN: 978-3-030-28346-9. DOI: 10.1007/978-3-030-28346-9_8 (cit. on p. 139).

Koopman, P., R. Plasmeijer and P. Achten (2011). 'An Executable and Testable Semantics for iTasks'. In: *Implementation and Application of Functional Languages*. Ed. by S.-B. Scholz and O. Chitil. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 212–232. ISBN: 978-3-642-24452-0 (cit. on p. 137).

Krishnamurthi, S. (2001). 'Linguistic reuse'. PhD thesis. Houston, USA: Rice University. 119 pp. (cit. on pp. 6, 77).

Krishnamurthi, S., M. Felleisen and D. P. Friedman (1998). 'Synthesizing object-oriented and functional design to promote re-use'. In: *ECOOP'98 — Object-Oriented Programming*. Ed. by E. Jul. event-place: Brussels, Belgium. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 91–113. ISBN: 978-3-540-69064-1 (cit. on p. 20).

Lämmel, R. and S. Peyton Jones (2003). 'Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming'. In: *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*. TLDI '03. event-place: New Orleans, Louisiana, USA. New York, NY, USA: ACM, pp. 26–37. ISBN: 1-58113-649-8. DOI: 10.1145/604174.604179 (cit. on p. 58).

Läufer, K. (1994). 'Combining type classes and existential types'. In: *Proceedings of the Latin American Informatic Conference (PANEL)*. Latin American Informatics Conferenc. event-place: Monterrey, Mexico. ITESM (cit. on p. 24).

Läufer, K. (1996). 'Type classes with existential types'. In: *Journal of Functional Programming* 6.3. Publisher: Cambridge University Press, pp. 485–518. DOI: 10.1017/S0956796800001817 (cit. on p. 24).

Lee, B., B. Wiedermann, M. Hirzel, R. Grimm and K. S. McKinley (2010). 'Jinn: Synthesizing Dynamic Bug Detectors for Foreign Language Interfaces'. In: *SIGPLAN Not.* 45.6, pp. 36–49. ISSN: 0362-1340. DOI: 10.1145/1809028.1806601 (cit. on p. 173).

Lee, J. K., S. J. Jung, S. D. Kim, W. H. Jang and D. H. Ham (2001). 'Component identification method with coupling and cohesion'. In: *Proceedings Eighth Asia-Pacific Software Engineering Conference*. IEEE. Macao, China: IEEE, pp. 79–86 (cit. on p. 148).

Leijen, D. and E. Meijer (2000). 'Domain Specific Embedded Compilers'. In: *Proceedings of the 2nd Conference on Domain-Specific Languages*. DSL '99. event-place: Austin, Texas, USA. New York, NY, USA: ACM, pp. 109–122. ISBN: 1-58113-255-7. DOI: 10.1145/331960.331977 (cit. on p. 44).

Leijen, D. and E. Meijer (2001). *Parsec: Direct Style Monadic Parser Combinators For The Real World*. UU-CS-2001-27. Utrecht: Universiteit Utrecht, p. 22 (cit. on p. 56).

Levis, P. and D. Culler (2002). 'Maté: A tiny virtual machine for sensor networks'. In: *ACM Sigplan Notices* 37.10, pp. 85–95 (cit. on pp. 133, 152).

Lewis, P. T. (1985). 'Speech'. Speech. Speech. Congressional Black Caucus Foundation 15th Annual Legislative Weekend. Washington, D.C. URL: http://www.chetansharma.com/correcting-the-iot-history/ (cit. on p. 3).

Light, R. (2017). 'Mosquitto: server and client implementation of the MQTT protocol'. In: *Journal of Open Source Software* 2.13, p. 265 (cit. on p. 170).

Lijnse, B. (2022). *Toppyt*. URL: https://gitlab.com/baslijnse/toppyt (visited on 07/10/2022) (cit. on p. 12).

Lijnse, B., J. M. Jansen, R. Plasmeijer et al. (2012). 'Incidone: A task-oriented incident coordination tool'. In: *Proceedings of the 9th International Conference on Information Systems for Crisis Response and Management, ISCRAM*. Vol. 12 (cit. on p. 135).

Lijnse, B., R. Nanne, J. M. Jansen and R. Plasmeijer (2011). 'Capturing the Netherlands Coast Guard's SAR Workflow with iTasks'. In: *Proceedings of the 8th International ISCRAM Conference*. Lisbon, Portugal, p. 10 (cit. on p. 135).

Lilis, Y. and A. Savidis (2019). 'A Survey of Metaprogramming Languages'. In: *ACM Comput. Surv.* 52.6. Place: New York, NY, USA Publisher: ACM. ISSN: 0360-0300. DOI: 10.1145/3354584 (cit. on p. 46).

Löh, A. and R. Hinze (2006). 'Open Data Types and Open Functions'. In: *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP '06. event-place: Venice, Italy. New York, NY, USA: ACM, pp. 133–144. ISBN: 1-59593-388-3. DOI: 10.1145/1140335.1140352 (cit. on pp. 32, 61).

Lubbers, M., P. Koopman and R. Plasmeijer (2019). 'Multitasking on Microcontrollers using Task Oriented Programming'. In: *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. COnference on COmposability, COmprehensibility and COrrectness of Working Software. Opatija, Croatia, pp. 1587–1592. DOI: 10.23919/MIPRO.2019.8756711 (cit. on pp. 14, 139, 178).

Lubbers, M. (2017). 'Task Oriented Programming and the Internet of Things'. Master's Thesis. Nijmegen: Radboud University. 69 pp. (cit. on pp. 13, 139).

Lubbers, M. (2022a). 'Deep Embedding with Class'. In: *Trends in Functional Programming*. Ed. by W. Swierstra and N. Wu. Cham: Springer International Publishing, pp. 39–58. ISBN: 978-3-031-21314-4. DOI: 10.1007/978-3-031-21314-4_5 (cit. on p. 13).

Lubbers, M. (2022b). *hTask*. URL: https://gitlab.com/mlubbers/hTask (visited on 07/10/2022) (cit. on p. 12).

Lubbers, M. and P. Koopman (2022). 'Green Computing for the Internet of Things'. In: *SusTrainable Summer School 2022, Rijeka, Croatia, July 4–5, 2022, Revised Selected Papers*. in-press. Cham: Springer International Publishing, p. 1 (cit. on pp. 14, 139).

Lubbers, M., P. Koopman and R. Plasmeijer (2018). 'Task Oriented Programming and the Internet of Things'. In: *Proceedings of the 30th Symposium on the Implementation and Application of Functional Programming Languages*. International Symposium on Implementation and Application of Functional Languages. Lowell, MA: ACM, p. 12. ISBN: 978-1-4503-7143-8. DOI: 10.1145/3310232.3310239 (cit. on pp. 13, 139).

Lubbers, M., P. Koopman and R. Plasmeijer (2021). 'Interpreting Task Oriented Programs on Tiny Computers'. In: *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*. Symposium on Implementation and Application of Functional Languages. Ed. by J. Stutterheim and W. N. Chin. IFL '19. event-place: Singapore, Singapore. New York, NY, USA: ACM. ISBN: 978-1-4503-7562-7. DOI: 10.1145/3412932.3412936 (cit. on pp. 14, 139, 144).

Lubbers, M., P. Koopman and R. Plasmeijer (2023a). 'First-Class Data Types in Shallow Embedded Domain-Specific Languages using Metaprogramming'. In: *Proceedings of the 34st Symposium on Implementation and Application of Functional Languages*. Symposium on Implementation and Application of Functional Languages. IFL '22. event-place: Copenhagen, Denmark. in-press. New York, NY, USA: ACM. ISBN: 978-1-4503-9831-2. DOI: 10.1145/3587216.3587219 (cit. on p. 13).

Lubbers, M., P. Koopman and R. Plasmeijer (2023b). 'Writing Internet of Things Applications with Task Oriented Programming'. In: *Composability, Comprehensibility and Correctness of Working Software, 8th Summer School, Budapest, Hungary, June 17–21, 2019, Revised Selected Papers*. Lecture Notes in Computer Science 11950. in-press. preprint at: https://arxiv.org/abs/2212.04193. Cham: Springer, p. 51 (cit. on pp. 14, 139, 178).

Lubbers, M., P. Koopman, A. Ramsingh, J. Singer and P. Trinder (2020). 'Tiered versus Tierless IoT Stacks: Comparing Smart Campus Software Architectures'. In: *Proceedings of the 10th International Conference on the Internet of Things*. 10th International Conference on the Internet of Things. IoT '20. event-place: Malmö, Sweden. Malmö: ACM. ISBN: 978-1-4503-8758-3. DOI: 10.1145/3410992.3411002 (cit. on pp. 15, 140, 145).

Lubbers, M., P. Koopman, A. Ramsingh, J. Singer and P. Trinder (2023c). 'Could Tierless Languages Reduce IoT Development Grief?' In: *ACM Trans. Internet Things* 4.1. Place: New York, NY, USA Publisher: ACM. ISSN: 2691-1914. DOI: 10.1145/3572901 (cit. on pp. 15, 140).

Lynagh, I. (2003). *Unrolling and Simplifying Expressions with Template Haskell*. URL: http://web.comlab.ox.ac.uk/oucl/work/ian.lynagh/papers/ (visited on 07/09/2021) (cit. on p. 59).

MacCormack, A., J. Rusnak and C. Y. Baldwin (2007). 'The impact of component modularity on design evolution: Evidence from the software industry'. In: *Harvard Business School Technology & Operations Mgt. Unit Research Paper* 08.038. DOI: 10.2139/ssrn.1071720 (cit. on p. 148).

Mainland, G. (2007). 'Why It's Nice to Be Quoted: Quasiquoting for Haskell'. In: *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*. Haskell '07. event-place: Freiburg, Germany. New York, NY, USA: ACM, pp. 73–82. ISBN: 978-1-59593-674-5. DOI: 10.1145/1291201.1291211 (cit. on pp. 55, 60).

Materzok, M. (2022). 'Generating Circuits with Generators'. In: *Proc. ACM Program. Lang.* 6 (ICFP). Place: New York, NY, USA Publisher: ACM. DOI: 10.1145/3549821 (cit. on p. 59).

Mayer, P. and A. Bauer (2015). 'An Empirical Analysis of the Utilization of Multiple Programming Languages in Open Source Projects'. In: *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*. EASE '15. Nanjing, China: ACM. ISBN: 9781450333504. DOI: 10.1145/2745802.2745805 (cit. on p. 169).

Mayer, P., M. Kirsch and M. A. Le (2017). 'On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers'. In: *Journal of Software Engineering Research and Development* 5.1, p. 1 (cit. on p. 149).

Mazzei, D., G. Baldi, G. Montelisciani and G. Fantoni (2018). 'A full stack for quick prototyping of IoT solutions'. In: *Annals of Telecommunications* 73.7-8, pp. 439–449 (cit. on p. 148).

McDonell, T. L., J. D. Meredith and G. Keller (2022). 'Embedded Pattern Matching'. In: *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*. Haskell 2022. event-place: Ljubljana, Slovenia. New York, NY, USA: ACM, pp. 123–136. ISBN: 978-1-4503-9438-3. DOI: 10.1145/3546189.3549917 (cit. on p. 58).

MicropythonTeam (2022). *MicroPython Differences from CPython*. URL: https://docs.micropython.org/en/latest/genrst/index.html (cit. on p. 165).

Mitchell, J. C. and G. D. Plotkin (1988). 'Abstract Types Have Existential Type'. In: *ACM Trans. Program. Lang. Syst.* 10.3. Place: New York, NY, USA Publisher: ACM, pp. 470–502. ISSN: 0164-0925. DOI: 10.1145/44501.45065 (cit. on p. 24).

Motta, R. C., K. M. de Oliveira and G. H. Travassos (2018). 'On Challenges in Engineering IoT Software Systems'. In: *Proceedings of the XXXII Brazilian Symposium on Software Engineering*. SBES '18. Sao Carlos, Brazil: ACM, pp. 42–51. ISBN: 9781450365031. DOI: 10.1145/3266237.3266263 (cit. on p. 149).

Muccini, H. and M. T. Moghaddam (2018). 'IoT Architectural Styles'. In: *Software Architecture*. Ed. by C. E. Cuesta, D. Garlan and J. Pérez. Cham: Springer International Publishing, pp. 68–85. ISBN: 978-3-030-00761-4 (cit. on p. 146).

Najd, S., S. Lindley, J. Svenningsson and P. Wadler (2016). 'Everything Old is New Again: Quoted Domain-Specific Languages'. In: *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. PEPM '16. event-place: St. Petersburg, FL, USA. New York, NY, USA: ACM, pp. 25–36. ISBN: 978-1-4503-4097-7. DOI: 10.1145/2847538.2847541 (cit. on p. 60).

Najd, S. and S. Peyton Jones (2017). 'Trees that Grow'. In: *Journal of Universal Computer Science* 23.1, pp. 42–62 (cit. on p. 32).

Nelson, T., A. D. Ferguson, M. J. G. Scheer and S. Krishnamurthi (2014). 'Tierless Programming and Reasoning for Software-Defined Networks'. In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, pp. 519–531. ISBN: 978-1-931971-09-6. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/nelson (cit. on p. 132).

Nilsson, H., A. Courtney and J. Peterson (2002). 'Functional Reactive Programming, Continued'. In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. Haskell '02. event-place: Pittsburgh, Pennsylvania. New York, NY, USA: ACM, pp. 51–64. ISBN: 1-58113-605-6. DOI: 10.1145/581690.581695 (cit. on pp. 134, 150).

Nižetić, S., P. Šolić, D. L.-I. González-de-Artaza and L. Patrono (2020). 'Internet of Things (IoT): Opportunities, issues and challenges towards a smart and sustainable future'. In: *Journal of Cleaner Production* 274, p. 122877. ISSN: 0959-6526. DOI: https://doi.org/10.1016/j.jclepro.2020.122877 (cit. on p. 117).

Nöcker, E. G. J. M. H., J. E. W. Smetsers, M. C. J. D. van Eekelen and M. J. Plasmeijer (1991). 'Concurrent clean'. In: *PARLE '91 Parallel Architectures and Languages Europe*. Ed. by E. H. L. Aarts, J. van Leeuwen and M. Rem. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 202–219. ISBN: 978-3-540-47472-2 (cit. on p. 187).

Norell, U. and P. Jansson (2004). 'Prototyping Generic Programming in Template Haskell'. In: *Mathematics of Program Construction*. Ed. by D. Kozen. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 314–333. ISBN: 978-3-540-27764-4 (cit. on p. 59).

O'Donnell, J. T. (2004). 'Embedding a Hardware Description Language in Template Haskell'. In: *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*. Ed. by C. Lengauer, D. Batory, C. Consel and M. Odersky. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 143–164. ISBN: 978-3-540-25935-0. DOI: 10.1007/978-3-540-25935-0_9 (cit. on p. 59).

Odersky, M. and K. Läufer (1996). 'Putting Type Annotations to Work'. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96. event-place: St. Petersburg Beach, Florida, USA. New York, NY, USA: ACM, pp. 54–67. ISBN: 0-89791-769-3. DOI: 10.1145/237721.237729 (cit. on pp. 31, 76).

Omar, C., I. Voysey, M. Hilton, J. Aldrich and M. A. Hammer (2017). 'Hazelnut: A Bidirectionally Typed Structure Editor Calculus'. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL '17. event-place: Paris, France. New York, NY, USA: ACM, pp. 86–99. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009900 (cit. on p. 138).

Oortgiese, A., J. van Groningen, P. Achten and R. Plasmeijer (2017). 'A Distributed Dynamic Architecture for Task Oriented Programming'. In: *Proceedings of the 29th Symposium on Implementation and Application of Functional Programming Languages*. Bristol, UK: ACM, p. 7 (cit. on pp. 152, 154, 163).

Peyton Jones, S. (1987). *The Implementation of Functional Programming Languages*. Hertfordshire: Prentice Hall. URL: https://www.microsoft.com/en-us/research/publication/the-implementation-of-functional-programming-languages/ (cit. on p. 56).

Peyton Jones, S., ed. (2003). *Haskell 98 language and libraries: the revised report*. Cambridge: Cambridge University Press. 270 pp. ISBN: 0-521 826144 (cit. on pp. 20, 22, 24, 43, 44).

Peyton Jones, S. and P. Wadler (1993). 'Imperative Functional Programming'. In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '93. event-place: Charleston, South Carolina, USA. New York, NY, USA: ACM, pp. 71–84. ISBN: 0-89791-560-7. DOI: 10.1145/158511.158524 (cit. on pp. 154, 188).

Pfenning, F. and C. Elliott (1988). 'Higher-Order Abstract Syntax'. In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI '88. event-place: Atlanta, Georgia, USA. New York, NY, USA: ACM, pp. 199–208. ISBN: 0-89791-269-1. DOI: 10.1145/53990.54010 (cit. on pp. 44, 78).

Philips, L., C. de Roover, T. van Cutsem and W. de Meuter (2014). 'Towards Tierless Web Development without Tierless Languages'. In: *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2014. Portland, Oregon, USA: ACM, pp. 69–81. ISBN: 9781450332101. DOI: 10.1145/2661136.2661146 (cit. on pp. 151, 180).

Pickering, M., A. Löh and N. Wu (2020). 'Staged Sums of Products'. In: *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*. Haskell 2020. event-place: Virtual Event, USA. New York, NY, USA: ACM, pp. 122–135. ISBN: 978-1-4503-8050-8. DOI: 10.1145/3406088.3409021 (cit. on p. 60).

Pickering, M., N. Wu and C. Kiss (2019). 'Multi-Stage Programs in Context'. In: *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*. Haskell 2019. event-place: Berlin, Germany. New York, NY, USA: ACM, pp. 71–84. ISBN: 978-1-4503-6813-1. DOI: 10.1145/3331545.3342597 (cit. on p. 60).

Piers, J. (2016). 'Task-Oriented Programming for developing non-distributed interruptible embedded systems'. Master's Thesis. Nijmegen: Radboud University. 81 pp. (cit. on pp. 12, 135).

Plamauer, S. and M. Langer (2017). 'Evaluation of micropython as application layer programming language on cubesats'. In: *ARCS 2017; 30th International Conference on Architecture of Computing Systems*. VDE. Vienna, Austria: VDE, pp. 1–9 (cit. on p. 165).

Plasmeijer, R. and P. Achten (2006). 'A conference management system based on the iData toolkit'. In: *Symposium on Implementation and Application of Functional Languages*. Springer, pp. 108–125 (cit. on p. 135).

Plasmeijer, R., P. Achten and P. Koopman (2007a). 'iTasks: Executable Specifications of Interactive Work Flow Systems for the Web'. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*. Freiburg, Germany: ACM, pp. 141–152. ISBN: 978-1-59593-815-2 (cit. on p. 154).

Plasmeijer, R., P. Achten and P. Koopman (2007b). 'iTasks: executable specifications of interactive work flow systems for the web'. In: *ACM SIGPLAN Notices* 42.9, pp. 141–152 (cit. on p. 8).

Plasmeijer, R., M. van Eekelen and J. van Groningen (2021). *Clean Language Report version 3.1*. Nijmegen: Institute for Computing and Information Sciences, p. 127. (Visited on 22/12/2021) (cit. on pp. 75, 76, 187, 189).

Plasmeijer, R. and P. Koopman (2016). 'A Shallow Embedded Type Safe Extendable DSL for the Arduino'. In: *Trends in Functional Programming*. Vol. 9547. Lecture Notes in Computer Science. Cham: Springer International Publishing. ISBN: 978-3-319-39110-6. DOI: 10.1007/978-3-319-39110-6. (Visited on 22/02/2017) (cit. on p. 139).

Plasmeijer, R., B. Lijnse, S. Michels, P. Achten and P. Koopman (2012). 'Task-Oriented Programming in a Pure Functional Language'. In: *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming*. PPDP '12. event-place: Leuven, Belgium. New York, NY, USA: ACM, pp. 195–206. ISBN: 978-1-4503-1522-7. DOI: 10.1145/2370776.2370801 (cit. on pp. 6, 8, 137, 154, 180).

Polak, G. and J. Jarosz (2006). 'Automatic Graphical User Interface Form Generation Using Template Haskell'. In: *Revised Selected Papers from the Seventh Symposium on Trends in Functional Programming, TFP 2006, Nottingham, United Kingdom, 19-21 April 2006*. Ed. by H. Nilsson. Vol. 7. Trends in Functional Programming. event-place: Nottingham, UK. Bristol, UK: Intellect, pp. 1–11. ISBN: 978-1-84150-188-8 (cit. on p. 59).

Ravulavaru, A. (2018). *Enterprise internet of things handbook : build end-to-end IoT solutions using popular IoT platforms*. eng. Birmingham, UK: Packt Publishing. ISBN: 1-78883-378-3 (cit. on p. 144).

Reynolds, J. C. (1978). 'User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction'. In: *Programming Methodology: A Collection of Articles by Members of IFIP WG2.3*. Ed. by D. Gries. New York, NY: Springer New York, pp. 309–317. ISBN: 978-1-4612-6315-9. DOI: 10.1007/978-1-4612-6315-9_22 (cit. on p. 20).

Rhiger, M. (2009). 'Type-safe pattern combinators'. In: *Journal of Functional Programming* 19.2. Publisher: Cambridge University Press, pp. 145–156. DOI: 10.1017/S0956796808007089 (cit. on p. 58).

Rodriguez, A., J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov and B. C. d. S. Oliveira (2008). 'Comparing Libraries for Generic Programming in Haskell'. In: *Proceedings of the First ACM SIGPLAN Symposium on Haskell*. Haskell '08. Victoria, BC, Canada: ACM, pp. 111–122. ISBN: 9781605580647. DOI: 10.1145/1411286.1411301 (cit. on p. 154).

de Roos, C. (2020). *Custom Type Errors for Class-Based Embedded DSLs*. Research Intership Report. Radboud University (cit. on p. 138).

Rosenberg, J. (1997). 'Some misconceptions about lines of code'. In: *Proceedings fourth international software metrics symposium*. IEEE. Albuquerque, NM, USA: IEEE, pp. 137–142. DOI: 10.1109/METRIC.1997.637174 (cit. on pp. 145, 166).

Sanchez-Iborra, R. and A. F. Skarmeta (2020). 'TinyML-Enabled Frugal Smart Objects: Challenges and Opportunities'. In: *IEEE Circuits and Systems Magazine* 20.3, pp. 4–18. DOI: 10.1109/MCAS.2020.3005467 (cit. on p. 136).

Sant'Anna, F., N. Rodriguez, R. Ierusalimschy, O. Landsiedel and P. Tsigas (2013). 'Safe system-level concurrency on resource-constrained nodes'. In: *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*. ACM, p. 11 (cit. on p. 134).

Sarkar, A. and M. Sheeran (2020). 'Hailstorm: A Statically-Typed, Purely Functional Language for IoT Applications'. In: *Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming*. PPDP '20. event-place: Bologna, Italy. New York, NY, USA: ACM. ISBN: 978-1-4503-8821-4. DOI: 10.1145/3414080.3414092 (cit. on pp. 135, 150).

Sawada, K. and T. Watanabe (2016). 'Emfrp: a functional reactive programming language for small-scale embedded systems'. In: *Companion Proceedings of the 15th International Conference on Modularity*. ACM, pp. 36–44 (cit. on pp. 135, 150).

Seefried, S., M. Chakravarty and G. Keller (2004). 'Optimising Embedded DSLs Using Template Haskell'. In: *Generative Programming and Component Engineering*. Ed. by G. Karsai and E. Visser. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 186–205. ISBN: 978-3-540-30175-2 (cit. on p. 60).

Serrano, A. (2018). 'Type Error Customization for Embedded Domain-Specific Languages'. PhD Thesis. Utrecht University (cit. on p. 138).

Serrano, M., E. Gallesio and F. Loitsch (2006). 'Hop: a language for programming the web 2.0'. In: *OOPSLA Companion*. Portland, Oregon, USA: ACM, pp. 975–985 (cit. on pp. 144, 149, 174, 180).

Sethi, P. and S. R. Sarangi (2017). 'Internet of things: architectures, protocols, and applications'. In: *Journal of Electrical and Computer Engineering* 2017 (cit. on pp. 144, 146).

Sheard, T. (2001). 'Accomplishments and Research Challenges in Meta-programming'. In: *Semantics, Applications, and Implementation of Program Generation*. Ed. by W. Taha. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 2–44. ISBN: 978-3-540-44806-8 (cit. on pp. 6, 46).

Sheard, T. and S. Peyton Jones (2002). 'Template Meta-Programming for Haskell'. In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. Haskell '02. event-place: Pittsburgh, Pennsylvania. New York, NY, USA: ACM, pp. 1–16. ISBN: 1-58113-605-6. DOI: 10.1145/581690.581691 (cit. on pp. 46, 59).

Sheetz, S. D., D. Henderson and L. Wallace (2009). 'Understanding developer and manager perceptions of function points and source lines of code'. In: *Journal of Systems and Software* 82.9, pp. 1540–1549 (cit. on p. 166).

Shi, W., J. Cao, Q. Zhang, Y. Li and L. Xu (2016). 'Edge Computing: Vision and Challenges'. In: *IEEE Internet of Things Journal* 3.5, pp. 637–646. DOI: 10.1109/JIOT.2016.2579198 (cit. on p. 136).

Shibanai, K. and T. Watanabe (2018). 'Distributed Functional Reactive Programming on Actor-Based Runtime'. In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. AGERE 2018. event-place: Boston, MA, USA. New York, NY, USA: ACM, pp. 13–22. ISBN: 978-1-4503-6066-1. DOI: 10.1145/3281366.3281370 (cit. on pp. 135, 150, 151, 153).

Shioda, M., H. Iwasaki and S. Sato (2014). 'LibDSL: A Library for Developing Embedded Domain Specific Languages in d via Template Metaprogramming'. In: *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*. GPCE 2014. event-place: Västerås, Sweden. New York, NY, USA: ACM, pp. 63–72. ISBN: 978-1-4503-3161-6. DOI: 10.1145/2658761.2658770 (cit. on p. 59).

Sivieri, A., L. Mottola and G. Cugola (2012). 'Drop the phone and talk to the physical world: Programming the internet of things with Erlang'. In: *2012 Third International Workshop on Software Engineering for Sensor Network Applications (SESENA)*. IEEE, pp. 8–14 (cit. on pp. 151, 153).

Staps, C., J. van Groningen and R. Plasmeijer (2019). 'Lazy Interworking of Compiled and Interpreted Code for Sandboxing and Distributed Systems'. In: *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*. IFL '19. event-place: Singapore, Singapore. New York, NY, USA: ACM. ISBN: 978-1-4503-7562-7. DOI: 10.1145/3412932.3412941 (cit. on p. 8).

Steenvoorden, T. (2022). *TopHat: Task-Oriented Programming with Style*. Nijmegen: UB Nijmegen. ISBN: 978-94-6458-595-7 (cit. on p. 12).

Steenvoorden, T., N. Naus and M. Klinik (2019). 'TopHat: A Formal Foundation for Task-Oriented Programming'. In: *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*. PPDP '19. event-place: Porto, Portugal. New York, NY, USA: ACM. ISBN: 978-1-4503-7249-7. DOI: 10.1145/3354166.3354182 (cit. on pp. 12, 137, 180).

Steiner, H.-C. (2009). 'Firmata: Towards Making Microcontrollers Act Like Extensions of the Computer.' In: *NIME*, pp. 125–130 (cit. on p. 132).

Strack, I. (2015). *Getting Started with Meteor.js JavaScript Framework*. Packt Publishing Ltd (cit. on p. 149).

Stutterheim, J., P. Achten and R. Plasmeijer (2018). 'Maintaining Separation of Concerns Through Task Oriented Software Development'. In: *Trends in Functional Programming*. Ed. by M. Wang and S. Owens. Vol. 10788. Cham: Springer International Publishing, pp. 19–38. ISBN: 978-3-319-89718-9. DOI: 10.1007/978-3-319-89719-6. (Visited on 14/01/2019) (cit. on pp. 6, 7, 134, 150, 170).

Suchocki, R. and S. Kalvala (2015). 'Microscheme: Functional programming for the Arduino'. In: *Proceedings of the 2014 Scheme and Functional Programming Workshop*. Scheme and Functional Programming Workshop. CS Techreport 718. Washington DC, USA: University of Indiana, p. 9 (cit. on pp. 133, 176).

Sun, Y., U. Dhandhania and B. C. d. S. Oliveira (2022). 'Compositional Embeddings of Domain-Specific Languages'. In: *Proc. ACM Program. Lang.* 6 (OOPSLA2), p. 34. DOI: 10.1145/3563294 (cit. on pp. 33, 34, 37).

Suzuki, K., K. Nagayama, K. Sawada and T. Watanabe (2017). 'CFRP: A Functional Reactive Programming Language for Small-Scale Embedded Systems'. en. In: *Theory and Practice of Computation*. The University of The Philippines Cebu, Cebu City, The Philippines: WORLD SCIENTIFIC, pp. 1–13. ISBN: 978-981-323-406-2. DOI: 10.1142/9789813234079_0001. (Visited on 02/03/2022) (cit. on pp. 135, 150).

Svenningsson, J. and E. Axelsson (2013). 'Combining Deep and Shallow Embedding for EDSL'. In: *Trends in Functional Programming*. Ed. by H.-W. Loidl and R. Peña. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 21–36. ISBN: 978-3-642-40447-4. DOI: 10.1007/978-3-642-40447-4_2 (cit. on p. 33).

Swierstra, W. (2008). 'Data types à la carte'. In: *Journal of functional programming* 18.4, pp. 423–436. DOI: 10.1017/S0956796808006758 (cit. on pp. 32, 61).

BIBLIOGRAPHY

213

Tanganelli, G., C. Vallati and E. Mingozzi (2015). 'CoAPthon: Easy development of CoAP-based IoT applications with Python'. In: *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. Milan, Italy: IEEE, pp. 63–68 (cit. on p. 175).

Terei, D., S. Marlow, S. Peyton Jones and D. Mazières (2012). 'Safe Haskell'. In: *Proceedings of the 2012 Haskell Symposium*. Haskell '12. event-place: Copenhagen, Denmark. New York, NY, USA: ACM, pp. 137–148. ISBN: 978-1-4503-1574-6. DOI: 10.1145/2364506.2364524 (cit. on p. 47).

TOP Software (2023). *VIIA (Vessel Information Integrating Application)*. URL: https://www.top-software.nl/VIIA.html (visited on 06/02/2023) (cit. on p. 8).

Torrano, C. and C. Segura (2005). 'Strictness Analysis and let-to-case Transformation using Template Haskell'. In: *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005*. Ed. by M. C. J. D. v. Eekelen. Vol. 6. Trends in Functional Programming. event-place: Talinn, Estonia. Bristol, UK: Intellect, pp. 429–442. ISBN: 978-1-84150-176-5 (cit. on p. 60).

Transforma Insights (2023). *Current IoT Forecast Highlights*. accessed-on: 2023-01-19. Transforma Insights. URL: https://transformainsights.com/research/forecast/highlights (visited on 16/01/2023) (cit. on p. 1).

Tratt, L. (2008). 'Domain Specific Language Implementation via Compile-Time Meta-Programming'. In: *ACM Trans. Program. Lang. Syst.* 30.6. Place: New York, NY, USA Publisher: ACM. ISSN: 0164-0925. DOI: 10.1145/1391956.1391958 (cit. on p. 6).

Troyer de, C., J. Nicolay and W. Meuter de (2018). 'Building IoT Systems Using Distributed First-Class Reactive Programming'. In: *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. Nicosia, Cyprus: IEEE, pp. 185–192 (cit. on pp. 135, 144, 149, 150).

Valliappan, N., R. Krook, A. Russo and K. Claessen (2020). 'Towards Secure IoT Programming in Haskell'. In: *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*. New York, NY, USA: ACM, pp. 136–150. ISBN: 978-1-4503-8050-8 (cit. on pp. 135, 150, 153, 180).

van der Veen, E. (2020). 'Mutable Collection Types in Shallow Embedded DSLs'. Master's Thesis. Nijmegen: Radboud University. 68 pp. (cit. on pp. 136, 139).

Verna, D. (2013). 'Extensible Languages: Blurring the Distinction between DSL and GPL'. In: *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*. Ed. by M. Mernik. Hershey, PA, USA: IGI Global, pp. 1–31. ISBN: 978-1-4666-2092-6. DOI: 10.4018/978-1-4666-2092-6.ch001 (cit. on p. 5).

Viera, M., F. Balestrieri and A. Pardo (2018). 'A Staged Embedding of Attribute Grammars in Haskell'. In: *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages*. IFL 2018. event-place: Lowell, MA, USA. New York, NY, USA: ACM, pp. 95–106. ISBN: 978-1-4503-7143-8. DOI: 10.1145/3310232.3310235 (cit. on p. 59).

de Vries, E. and A. Löh (2014). 'True Sums of Products'. In: *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming*. WGP '14. event-place: Gothenburg, Sweden. New York, NY, USA: ACM, pp. 83–94. ISBN: 978-1-4503-3042-8. DOI: 10.1145/2633628.2633634 (cit. on p. 60).

Wadler, P. (1998). *The expression problem*. E-mail. URL: https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt (visited on 24/02/2021) (cit. on p. 20).

Wand, M. (1980). 'Continuation-based multiprocessing'. In: *Proceedings of the 1980 ACM conference on LISP and functional programming - LFP '80*. the 1980 ACM conference. Stanford University, California, United States: ACM Press, pp. 19–28. DOI: 10.1145/800087.802786. (Visited on 13/02/2019) (cit. on p. 133).

Weisenburger, P., J. Wirth and G. Salvaneschi (2020). 'A survey of multitier programming'. In: *ACM Computing Surveys (CSUR)* 53.4, pp. 1–35 (cit. on pp. 149, 152, 153, 179, 180).

Wijkhuizen, M. (2018). 'Security analysis of the iTasks framework'. English. Bachelor's Thesis. Nijmegen: Radboud University. (Visited on 08/04/2017) (cit. on p. 153).

Wikipedia contributors (2022). *Rhapsody (music) — Wikipedia, The Free Encyclopedia*. In: *Wikipedia*. accessed on: 2022-09-06. URL: https://en.wikipedia.org/w/index.php?title=Rhapsody_(music)%5C&oldid=1068385257 (visited on 06/09/2022) (cit. on p. 2).

Willis, J., N. Wu and M. Pickering (2020). 'Staged Selective Parser Combinators'. In: *Proc. ACM Program. Lang.* 4 (ICFP). Place: New York, NY, USA Publisher: ACM. DOI: 10.1145/3409002 (cit. on p. 60).

Xie, N., M. Pickering, A. Löh, N. Wu, J. Yallop and M. Wang (2022). 'Staging with Class: A Specification for Typed Template Haskell'. In: *Proc. ACM Program. Lang.* 6 (POPL). Place: New York, NY, USA Publisher: ACM. DOI: 10.1145/3498723 (cit. on p. 60).

Yorgey, B. A., S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis and J. P. Magalhães (2012). 'Giving Haskell a Promotion'. In: *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*. TLDI '12. event-place: Philadelphia, Pennsylvania, USA. New York, NY, USA: ACM, pp. 53–66. ISBN: 978-1-4503-1120-5. DOI: 10.1145/2103786.2103795 (cit. on pp. 25, 35).

Young, D., M. Grebe and A. Gill (2021). 'On Adding Pattern Matching to Haskell-Based Deeply Embedded Domain Specific Languages'. In: *Practical Aspects of Declarative Languages: 23rd International Symposium, PADL 2021, Copenhagen, Denmark, January 18-19, 2021, Proceedings*. event-place: Copenhagen, Denmark. Berlin, Heidelberg: Springer-Verlag, pp. 20–36. ISBN: 978-3-030-67437-3. DOI: 10.1007/978-3-030-67438-0_2 (cit. on p. 58).

Zdancewic, S., L. Zheng, N. Nystrom and A. C. Myers (2002). 'Secure program partitioning'. In: *ACM Transactions on Computer Systems (TOCS)* 20.3, pp. 283–328 (cit. on p. 153).

Summary

The development of reliable software for the internet of things (IoT) is difficult because IoT systems are dynamic, interactive, distributed, collaborative, multi-tiered, and multitasking in nature. The complexity is increased further by semantic friction that arises through different hardware and software characteristics between tiers. Many computers that operate in IoT systems are *edge devices* that interact with the environment using sensors and actuators. Edge devices are often powered by low-cost microcontrollers designed for embedded applications. They have little memory, unhurried processors, and are slow in communication but are also small and energy efficient.

Task-oriented programming (TOP) can cope with the challenges of IoT programming. In TOP, the main building blocks are tasks, an abstract representation of work. During execution, the current value of the task is observable, and other tasks can act upon it. Collaboration patterns can be modelled by combining and transforming tasks into compound tasks. Programming edge devices benefits from TOP as well, but running such a system within the limitations of resource-constrained microcontrollers is not straightforward.

This dissertation demonstrates how to include edge devices in TOP systems using domain-specific languages (DSLs). With these techniques, all tiers and their interoperation of an IoT system are specified in a single high-level source, language, paradigm, high abstraction level, and type system. First, I present advanced DSL embedding techniques. Then mTask is shown, a TOP DSL for IoT edge devices, embedded in iTask. Tasks are constructed and compiled at run time in order to allow tasks to be tailored to the current work requirements. The task is then sent to the device for interpretation. A device is programmed once with a lightweight domain-specific OS to be used in an mTask system. This OS executes tasks in an energy-efficient way and automates all communications and data sharing. All aspects of the mTask system are shown: example applications, language design, implementation details, integration with iTask, and green computing facilities such as automatic sleeping.

Finally, tierless IoT programming is compared to traditional tiered programming. In tierless programming frameworks, the size of the code and the number of required programming languages is reduced significantly. By using a single paradigm and a system-wide type system, tierless programming reduces problems such as semantic friction; maintainability and robustness issues; and interoperation safety.

Samenvatting

Het ontwikkelen van betrouwbare software voor Internet of Things (IoT) systemen is moeilijk omdat ze dynamisch, interactief, gedistribueerd, samenwerkend, meerlaags en multitasking zijn. Ook draagt de semantische wrijving voorkomend uit de grote verscheidenheid aan hard- en software karakteristieken tussen de lagen bij aan dit probleem. Veel van deze computers zijn *randcomputers* die onderdeel zijn van het IoT. Randcomputers bestaan vaak uit goedkope microcontrollers, ontworpen voor geïntegreerde systemen, en ze interacteren met de buitenwereld door sensoren en actuatoren. Enerzijds hebben ze weinig geheugen, langzame rekenkernen en trage communicatie. Anderzijds zijn ze klein en hoogst energie-efficiënt.

Taakgeörienteerd programmeren (TOP) is geschikt om met de uitdagingen van IoT systemen om te gaan. In TOP zijn abstracte representaties van werk, de taken, de bouwstenen. Tijdens het uitvoeren van een taak kan de huidige waarde geobserveerd worden en hierop kunnen andere taken reageren. Door taken te combineren of te transformeren kunnen samenwerkingsvormen uitgedrukt worden. Van deze beschrijving wordt een computersysteem gegenereerd dat gebruikers begeleidt in het uitvoeren van het werk. Randcomputers hebben ook baat bij TOP, al is niet eenvoudig om TOP-systemen erop in te zetten.

Deze dissertatie laat zien hoe gehele IoT-systemen georkestreerd kunnen worden met TOP. Gebruik makend van techniek kunnen alle lagen van een IoT-systeem en hun samenwerking uitgedrukt worden in één hoog abstractieniveau, programmeertaal, paradigma en typesysteem. Allereerst laat ik enkele technieken zien om ingebedde domein-specifieke talen te maken. Daarna beschrijf ik mTask, een TOP-systeem voor randcomputers ingebed in iTask. Taken worden tijdens het uitvoeren opgebouwd, waardoor ze afgestemd kunnen worden op de huidige werkeisen. Vervolgens worden ze naar het apparaat gestuurd ter interpretatie. Na eenmalig uitgerust te worden met het domeinspecifieke besturingssysteem is een randcomputer geschikt voor mTask. Dit stuurprogramma voert de ontvangen taken energiezuinig uit en automatiseert tevens alle communicatie en dataverwerking. Alle aspecten van het mTask-systeem worden beschreven: voorbeeldprogramma's, taalontwerp, implementatiedetails, integratie met iTask en de energiezuinige functionaliteit.

Het laagloos programmeren van IoT systemen wordt ook vergeleken met traditioneel gelaagd programmeren. Laagloos programmeren leidt tot minder code en minder programmeertalen. Door het gebruik van één paradigma en een systeembreed typesysteem verlaagt laagloos programmeren de semantische wrijving, onderhouds- en robuustheidsproblematiek en moeizame onderlinge samenwerking.

Acknowledgements

While the research and writing carried out for this thesis was mostly done by me, it could not have been done without the support of many others.

First of all I would like to thank Rinus Plasmeijer, Pieter Koopman, and Jan Martin Jansen for the supervision, I learned a lot from you, not only regarding academia but in many other aspects of life as well. The BEST people, Adrian Ramsingh, Jeremy Singer, and Phil Trinder for the fruitful collaboration and memorable trip to Glasgow. The entire 3COWS/SusTrainable group, for offering a platform for the various summer schools I had the opportunity to teach; and not to mention the countless meetings, dinners, and drinks we had. The Royal Dutch Navy, in particular Teun de Groot and Ton van Heusden, for trusting me by funding the project. The manuscript committee, Sven-Bodo Scholz, Gabriele Keller, Mary Sheeran, for reading this work carefully.

All colleagues and others that I had the privilege of sharing an office with, meeting in conferences and summer schools, interact with in the department, or work with in some other way: Anett Fekete, Arjan Oortgiese, Bas Lijnse, Ellie Kimenai, Fok Bolderheij, Hans-Nikolai Vießmann, Ia Mgvdliashvili, Ingrid Berenbroek, John van Groningen, Jurriën Stutterheim, László Domoslai, Marie-José van Diem, Markus Klinik, Máté Cserép, Peter Achten, Ralf Hinze, Simone Meeuwsen, Sjaak Smetsers, Steffen Michels, Sven-Bodo Scholz, Tim Steenvoorden, and everyone else from department.

The many students that allowed and trusted me to (co) supervise them in their theses: Arjen Nederveen, Colin de Roos, Dave Artz, Elina Antonova, Erin van der Veen, Gijs Alberts, Haye Böhm, Matheus Amazonas Cabral de Andrade, Michel de Boer, Sjoerd Crooijmans, Willem de Vos.

I give special thanks to my mentors: Jos Baack for getting me to graduate high school, Francisco Torreira for sparking my love for academia, and Larry Caruthers for giving me valuable practical experience.

All friends that supported me in real life or through pen throughout the process: Pieter and Anouk; Chris and Maudy; Koen and Michelle; Alba; Александер; Annerieke; Camil and Devika; Emma; George; Jules and Nadia; Tim; and Truman.

Mijn ouders bedank ik voor hun tomeloze liefde en vertrouwen. Mijn oma, broers, schoonfamilie en alle andere familieleden die op wat voor manier dan ook bijgedragen hebben. Als laatste wil ik diegenen bedanken die het dichtst bij mij staan: Elvira, Rosalie en Liselotte. Bedankt voor jullie onmisbare en oneindige geduld, ondersteuning en liefde wanneer dat nodig was.

Research data management

This thesis research has been carried out under the research data management policy of the Institute for Computing and Information Science of the Radboud University, the Netherlands.[1]

The following research datasets have been produced during this PhD research:

- Chapter 2: Deep embedding with class:

  - Lubbers, M. (Radboud University) (2022): Literate Haskell/lhs2TEX source code for the paper "Deep Embedding with Class": TFP 2022. Zenodo. 10.5281/zenodo.6650880
  - Section 2.A: Reprise: reducing boilerplate:

    * Lubbers, M. (Radboud University) (2022): Library and examples for enhanced classy deep embedding. Zenodo. 10.5281/zenodo.7277498

- Chapter 3: First-class data types in shallow embedded domain-specific languages using metaprogramming:

  - Lubbers, M. (Radboud University); & Koopman, P. (Radboud University) (2022): Code for the paper "First-Class Data Types in Shallow Embedded Domain-Specific Languages using Metaprogramming": IFL 2022. Zenodo. 10.5281/zenodo.6416747

- Episode II: Orchestrating the Internet of Things using Task-Oriented Programming:

  - Lubbers, M. (Radboud University); Koopman, P. (Radboud University); & Plasmeijer, R. (Radboud University) (2020): Source code for the mTask language. DANS. 10.17026/dans-xx4-8zs9
  - Lubbers, M. (Radboud University); Koopman, P. (Radboud University); & Plasmeijer, R. (Radboud University) (2020): Source code for the multitasking mTask language integrated with the iTask system. DANS. 10.17026/dans-x2y-rtxx
  - Lubbers, M. (Radboud University); Koopman, P. (Radboud University); & Plasmeijer, R. (Radboud University) (2020): Source code for a simplified mTask language integrated with the iTask system. DANS. 10.17026/dans-xv6-fvxd

---

[1]https://www.ru.nl/icis/research-data-management/, accessed on: 20$^{\text{th}}$ January, 2020

- Lubbers, M. (Radboud University); Koopman, P. (Radboud University); & Plasmeijer, R. (Radboud University) (2021): Source code for the interpreted mTask language. DANS. 10.17026/dans-zrn-2wv3
- Crooijmans, S. (Radboud University); Lubbers, M. (Radboud University); & Koopman, P. (Radboud University) (2023): Code for the paper "Reducing the Power Consumption of IoT with Task-Oriented Programming": TFP 2022. Zenodo. 10.5281/zenodo.7634538
- Lubbers, M. (Radboud University); & Koopman, P. (Radboud University); Plasmeijer, R. (Radboud University) (2023): Code for the lecture notes: "Writing Internet of Things Applications with Task Oriented Programming". Zenodo. 10.5281/zenodo.7643284
- Lubbers, M. (Radboud University); & Koopman, P. (Radboud University) (2023): Code for the lecture notes: "Green Computing for the Internet of Things". Zenodo. 10.5281/zenodo.7643316

- Episode III: Tiered versus Tierless Programming:
  - Lubbers, M. (Radboud University); Koopman, P. (Radboud University); Ramsingh, A. (University of Glasgow); Singer, J. (University of Glasgow); & Trinder, P. (University of Glasgow) (2021): Source code, line counts and memory statistics for CRS, CWS, CRTS and CWTS. Zenodo. 10.5281/zenodo.5040754
  - Lubbers, M. (Radboud University); Koopman, P. (Radboud University); Ramsingh, A. (University of Glasgow); Singer, J. (University of Glasgow); & Trinder, P. (University of Glasgow) (2021): Source code, line counts and memory stats for PRS, PWS, PRT and PWT. Zenodo. 10.5281/zenodo.5081386
  - Lubbers, M. (Radboud University); Koopman, P. (Radboud University); Ramsingh, A. (University of Glasgow); Singer, J. (University of Glasgow); & Trinder, P. (University of Glasgow) (2021): Source code of the PRSS and CWSS applications. DANS. 10.17026/dans-zvf-4p9m

Curriculum Vitæ

Mart Lubbers

1992	Born May 27th, Oldenzaal
2004–2011	VWO at the Twents Carmelcollege De Thij, Oldenzaal
2011–2015	Bachelor's degree Artificial Intelligence at the Radboud University, Nijmegen
2013–2015	Research assistant at the Max Planck Institute for Psycholinguistics
2015–2017	Master's degree (cum laude) Computing Science (Software Science track) at the Radboud University, Nijmegen
2015–2017	Entrepreneur as IT Lubbers, Nijmegen
2017	Researcher at the Netherlands Defense Academy, Den Helder in the faculty of Military Sciences (fMIL).
2018	Junior researcher at the Radboud University, Nijmegen in the Institute for Computing and Information Sciences (iCIS).
2018–2023	PhD candidate at the Radboud University, Nijmegen in the Institute for Computing and Information Sciences (iCIS).

Glossary

I²C a simple serial communication protocol often used to connect sensors to microcontrollers

IoT internet of things

1-wire a simple single wire communication protocol often used to connect sensors to microcontrollers

3COWS the three "CO" (composability, comprehensibility, correctness) of working software

ABC Clean's intermediate graph-rewriting language

ADC analog-to-digital converter

ADT algebraic data type

API application programming interface

ARDSL Arduino DSL

Arduino a widely used framework for programming microcontrollers

AST abstract syntax tree

C a general-purpose imperative programming

C++ a general-purpose imperative programming language based on C

Clean a pure lazy functional programming language based on graph rewriting

CRS Clean Raspberry Pi system

CRTS Clean Raspberry Pi temperature sensor

CWS Clean WEMOS system

CWTS Clean WEMOS temperature sensor

DHT digital humidity and temperature

DSL domain-specific language

DVFS dynamic voltage and frequency scaling

eCO₂ CO_2 concentration calculated from TVOC measurements

eDSL embedded domain-specific language

FP functional programming

FPGA field-programmable gate array

FRP functional reactive programming

GADT generalised ADT

GHC Glasgow Haskell compiler

GPIO general-purpose input/output

GPL general-purpose language

GRS graph rewriting system

GUI graphical user interface

Haskell a pure lazy functional programming language designed by a committe as a concept language

HOAS high-order abstract syntax

I/O input/output

IDE integrated development environment

IR intermediate representation

ISR interrupt service routine

iTask a TOP eDSL for creating distributed multi-user collaborative web applications

JSON a open data interchange format using human readable text

LED light-emitting diode

MicroPython a Python implementation tailored for microcontrollers

MQTT a publish-subscribe network protocol designed for resource constrained devices

mTask a TOP eDSL for edge devices integrated with the iTask system

OLED organic LED
OS operating system

P-FRP priority-based FRP
PIR passive infrared
PRS Python Raspberry Pi system
PRTS Python Raspberry Pi temperature sensor
PWS MicroPython WEMOS system
PWTS MicroPython WEMOS temperature sensor
Python a multi-paradigm interpreted programming language

QDSL quoted DSL

RAM random-access memory
RFID radio-frequency identification

RTOS real-time OS
RTS run-time system

SDS shared data source
SLOC source lines of code
SPI a synchronous serial communication protocol often used to connect sensors to microcontrollers

TCP transmission control protocol
TH Template Haskell
TOP task-oriented programming
TopHat a TOP language designed to formally capture the essence of TOP
TOSD task-oriented software development
TTH typed Template Haskell
TVOC total volatile organic compounds

UI user interface
UoD universe of discourse
UoG University of Glasgow

VM virtual machine

WEMOS a popular ESP8266 microcontroller based prototyping platform supporting Arduino.
Wi-Fi is a family of wireless network protocols commonly used for local area networking