Master's Thesis

Computing Science

Developing Real Life, Task Oriented Applications for the Internet of Things

Matheus Amazonas Cabral de Andrade

September 26, 2018

Supervisor:
prof. dr. dr.h.c. ir. M.J. Plasmeijer

Daily Supervisor:
M. Lubbers MSc.

Second Reader:
dr. P.W.M. Koopman

Radboud University

Abstract

The Internet of Things is becoming ubiquitous. A growing number of objects are being equipped with devices that interact with its environment and exchange data via the Internet. Such devices are not powerful enough to run iTasks (an implementation of the Task Oriented Programming paradigm written in Clean) applications. The mTask Embedded Domain Specific Language was created with one goal in mind: to bring Internet of Things devices and the Task Oriented Programming paradigm together. But so far, only trivial applications were developed using mTask. This thesis assesses whether mTask can be used to develop real-life applications. Autohouse, the home automation application developed during the research, guided improvements in the mTask development environment and unearthed problems yet to be resolved.

Acknowledgements

I would like to thank everyone that helped me directly or indirectly in this adventure. Rinus and Mart for the supervision, weekly meetings, patience, advice and continuous feedback. Pieter for the help during the thesis and the feedback provided on this report. My friends around the globe that somehow overcame the distance and pushed me towards my goals. My Nijmegen friends for the daily advice, encouragement and occasional de-stressing parties. Last but not least, I would like to thank my family for the continuous support during these two last years. There is no doubt that I only made this far because of you. I love you. Especially, I would like to thank my parents for always trusting me and pointing me in the right direction.

Contents

Chapter 1

Introduction

1.1 Introduction

The Internet of Things (IoT) consists of a network of "things" (devices, computers, systems, etc.) that interact with each other via the Internet. These components can exchange data, monitor and manage each other. IoT is a global, growing phenomenon. According to Gartner, there were 3.96 billion connected "things" in 2016 and by 2020, 12.863 billion devices are expected to be connected to the Internet [10]. IoT has been used in a myriad of applications including home automation, fitness tracking, health care, warehouse monitoring, agriculture and industry manufacturing. IoT devices can be dedicated servers, personal computers, tablets, smartphones, smartwatches or compact devices operated by microcontrollers. Microcontrollers are small, cheap computers with limited resources and low power consumption commonly used to interface with the real world. They often gather data from sensors (movement, light, temperature, etc.), act on actuators (motors, LEDs, switches, etc.) and communicate with other devices.

Task Oriented Programming (TOP) is a new programming paradigm used to develop online, collaborative applications. Its central concept, a *task*, can be used to model different types of work performed both by users and systems. TOP provides a high level of abstraction, liberating the programmer from the burden of technical details, such as user interfaces. The iTasks [1] system implements TOP in the functional programming language Clean [3] as an Embedded Domain Specific Language (EDSL). It automatically generates as many assets as possible, turning it into a great tool for rapid prototyping. Given an iTasks program, the system automatically generates a web application that can be accessed via a web browser. Users can access this application to inspect and work on tasks. The system has been proved useful in many fields, including incident response operations and navy vessels automation [16, 6].

Although iTasks applications often require user interaction, some tasks could be automated. Examples are tasks that interact with the external world: reading room temperature, blinking an LED, detecting movement, unlocking a door, etc. The iTasks environment could benefit from such automation. Microcontrollers pose as great candidates to interface with the external world. They are affordable, energy efficient and are seamlessly combined with sensors and actuators. Unfortunately, — due to hardware limitations — microcontrollers are not suitable to run iTasks tasks. To bridge this gap, the mTask Domain Specific Language (DSL) was created to enable the execution of simple tasks on microcontrollers, bringing such devices to the iTasks world [14].

1.2 Research Question

Even though mTask was created to allow iTasks tasks to run on IoT devices, it has not been proved capable of running real-life applications yet. The examples built during its development were simple demonstrations and were far from real-life IoT applications. Given that, I propose the following research question:

Is it possible to develop real-life, IoT applications using mTask? If so, how can the development process be improved? If not, what are the challenges to solve to make it possible?

I plan to tackle the research question by example. Namely, trying to develop a real life IoT application using mTask. The attempt to develop such an application should display mTask's capability to create real life applications while displaying new opportunities to improve the development process.

The application I chose to develop tackles a popular problem in IoT: home automation. This application would be responsible for automating simple home management tasks such as turning the central heating system off when the room is warm, or opening up the curtains at a set time. The proposed application requires several IoT devices equipped with sensors (temperature, light, humidity, etc.) and actuators (LEDs, motors, relays, etc.) spread across rooms.

1.3 Report Structure

The remainder of this report is structured as follows. Chapter 2 quickly introduces DSLs and presents different strategies to build EDSLs. Chapter 3 introduces the concept of TOP and the iTasks system. Chapter 4 presents the mTask EDSL, which was the research focus. Chapter 5 describes the real-life application developed during research. Chapter 6 details the development process. Chapter 8 concludes presenting insight about the development process, answering the research question and proposing future research. Chapter 7 lists related work.

The source code for the real-life application developed during research can be accessed at `https://github.com/matheusamazonas/autohouse`.

The source code of the modified version of mTask used during research can be accessed at `https://gitlab.science.ru.nl/mlubbers/mTask/tree/peripherals`.

The source code of this report can be accessed at `https://github.com/matheusamazonas/masterthesis`.

Chapter 2

Embedded Domain Specific Languages

A General Purpose Language (GPL) is a computer language intended to be used in a wide range of domains. An example is the C++ programming language, which can be used in domains that vary from video games to web servers and systems programming. In contrast, a Domain Specific Language (DSL) is a computer language that was designed to be used in a particular domain. Game Maker Language, HTML, LaTeX and VHDL are examples of DSLs. These languages, when compared to GPLs, offer a higher level of abstraction from their target domain.

A DSL can be implemented using two different strategies: standalone and embedded. Each strategy has its advantages and disadvantages. In the former strategy, the language is built from the ground up, which consists of developing either a compiler or an interpreter. This strategy provides the language designer with a lot design freedom, but requires a cumbersome amount of work. In the latter strategy, the proposed DSL is embedded in another language. Therefore, the DSL inherits functionality from its host language. This inheritance often is desirable (e.g., a type checker) but sometimes might be undesirable (e.g., type coercion). This strategy frees its designer of building a new compiler.

The semantics of a Embedded Domain Specific Language (EDSL) are given by its views (also called backends). Common views are evaluation, pretty printing, compilation, optimization and verification. There are two main techniques for building an EDSL: deep and shallow embedding.

2.1 Deep Embedding

A deeply EDSL is represented in its host language as an Algebraic Data Type (ADT), where language constructs are modelled as data constructs. Views are functions that take the ADT as input and return another ADT that represents its semantics. An example of a simple deeply EDSL and its views (pretty printing and evaluation) can be seen on Listing 2.1.

```
:: MyDSL = I Int
    | B Bool
    | Add MyDSL MyDSL
    | Sub MyDSL MyDSL
    | And MyDSL MyDSL
    | Or MyDSL MyDSL
    | Var String
```

```
prettyPrint :: MyDSL → [String]
eval :: MyDSL → Int
```

Listing 2.1: A simple deeply EDSL and its views

The biggest advantage of deep embedding is that adding a view to the DSL is easy: simply create a function that transforms the ADT. One disadvantage is that extending the DSL might require a lot of work, since new code has to be created for every new construct in all the views. Another disadvantage is its lack of static type safety. As seen on Listing 2.1, `MyDSL` allows operations on mixed data types, such as addition on booleans and disjunction on integers.

Generalized Algebraic Data Types (GADTs) can be used to accomplish static type safety in deeply EDSLs, but Clean does not support them [5].

2.2 Shallow Embedding

Building a shallowly EDSL consists of representing the language constructs directly as its semantics. An example of a simple shallowly EDSL can be seen on Listing 2.2. In this example, the `Sem` ADT contains both the evaluation (`a`) and the pretty printing (`[String]`) views.

```
:: Sem a = Sem (a, [String])

add :: (Sem a)    (Sem a)    → Sem a    | + a
and :: (Sem Bool) (Sem Bool) → Sem Bool
eq ::  (Sem a)    (Sem a)    → Sem Bool | == a
var :: String → Sem Int
```

Listing 2.2: A simple shallowly EDSL

One advantage of shallow over deep embedding is that adding a new language construct is easy, given that each construct is just a function. Another advantage is that overloading can be achieved with the use of class constraints. In addition, static type checking is obtained without GADTs.

The biggest disadvantages of shallow embedding are based on the fact that all views are grouped in the same ADT. First, there is no separation of concerns. Second, there is computational waste when not all views are necessary. Finally, adding a new view in the semantics becomes increasingly burdensome. Another disadvantage of shallow embedding is that variables still remain unchecked during compilation. Finally, since the language constructs are functions, they can not be inspected as in deeply EDSLs.

2.2.1 Shallow Embedding with Type Classes

Type constructor classes can be used to avoid computing all the views of a shallowly EDSL [4]. A type constructor class containing the language constructs is created and each of the DSL views should provide an instance of the class. An example of a class-based shallowly EDSL can be seen on Listing 2.3.

```
:: Print a = P [String]
:: Eval a = E a

class myDSL v where
    lit :: t → v t | toString t
```

```
    add :: (v t)    (v t)    → v t    | + t
    and :: (v Bool) (v Bool) → v Bool
    eq  :: (v t)    (v t)    → v Bool | == t

instance myDSL Print where
    lit x = P [toString x]
    add (P x) (P y) = P (x ++ [" + ":y])
    and (P x) (P y) = P (x ++ [" && ":y])
    eq  (P x) (P y) = P (x ++ [" == ":y])

instance myDSL Eval where
    lit x = E x
    add (E x) (E y) = E (x + y)
    and (E x) (E y) = E (x && y)
    eq  (E x) (E y) = E (x == y)
```

Listing 2.3: A simple class-based shallowly EDSL

Class-based shallow embedding solves most of the problems previously faced with deep and shallow embedding. The language is statically typed, the views are separated, adding a view is simple, extending the language with a new construct is easy and operators can be overloaded.

Two problems remain. First, language constructs still can not be inspected. This is an inherent property of shallowly EDSLs and unfortunately cannot be eliminated.

Second, variables are still not checked at compile time. Functions can be used to solve this. This solution is not inherent to class-based deep embedding and can be used on regular deep embedding and on shallow embedding as well. Given that function arguments are well-typed in the host language, they can be used to represent typed variables. A variable declaration is defined as a function where its argument represents the variable and the function body represents the remaining of the program. As a result, any usage of the variable in the function body is well typed. In order to avoid constructs that are not variables on the left-hand side of the attribution operator, the types `Var` and `Expr` are introduced. In some views, these types are phantom types [15].

An example of a class-based shallowly EDSL with compile time variable checks can be seen on Listing 2.4. `Eval` is an example of a view with a phantom type: the type variable `b`.

```
:: Eval a b = E a
:: In a b = In infixl 0 a b
:: Var = Var
:: Expr = Expr

class expr v where
    lit :: t → v t Expr
    (+.) infixl 6 :: (v t p) (v t q) →  v t Expr | + t

class var v where
    var :: ((v t Var) → In t (v a p)) → v a p
    (=.) infixr 3 :: (v t Var) (v t p) → v t Expr

instance expr Eval where
    lit x = E x
    (+.) (E a) (E b) = E (a + b)
```

```
instance var Eval where
    var _ = ...
    (=.) _ _ = ...

test1 :: Eval Int Expr
test1 = var λk = 4 In
    k  =. k +. lit 7

test2 :: Eval Int Expr
test2 = var λk = 4 In
    k  =. lit True     // Compilation error
```

Listing 2.4: A simple class-based shallowly EDSL with compile time variable checks

As seen in the example above, the variable `k` can be used after its declaration in a well typed manner. Function `test1` compiles successfully, as expected. Function `test2`, in the other hand, does not compile due to a type error. Therefore, compile time check of variables was achieved.

Chapter 3

Task Oriented Programming

Task Oriented Programming (TOP) is a new programming paradigm used to develop online collaborative applications [1]. In a TOP application, users — both people and other systems — work together to accomplish a common goal [1]. Its central concept is called a task, an abstraction that can be used for many different types of work. This abstraction, along with other TOP concepts, allows the programmer to focus on design decisions instead of technical details. TOP programs are declarative: they focus on *what* work should be performed, rather than on *how* to perform it.

3.1 iTasks

The iTask system is an EDSL that implements the TOP paradigm. Its host language is the pure and lazy functional programming language *Clean* [3]. iTasks uses generic programming to automatically generate code for user-specified first-order data types whilst also allowing for specialization. Given an iTasks program, the system generates code for both the server and the client.

An iTasks task is a function that transforms a state, reacts to events and returns an observable value. A task is represented by `Task a` and its observable value by `TaskValue a`, where `a` is the type of the task. A `TaskValue` contains the current state of the value that a Task is processing. The possible states of a `TaskValue` are shown on Figure 3.1. As seen below, once a value stabilizes, it can not become unstable.

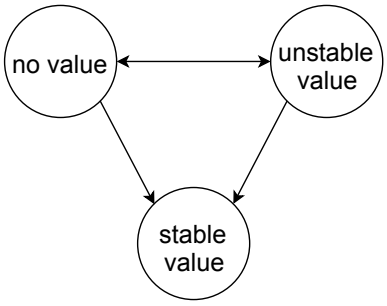


Figure 3.1: Possible states of a `TaskValue`

A Task can be of any type, as long as it provides instances for the type classes in the `iTask` type class collection. These instances can be automatically derived or explicitly declared. Automatic derivation is available for any first-order type as long as it is not abstract. Basic types have instances already defined in the iTasks library. Explicit declaration allows the user to define custom instances of the `iTask` class collection if the default derived instance is not suitable. Moreover, it allows instances for extendable, abstract and the function types.

3.2 Interaction

The iTasks library provides basic tasks for user interaction. Listing 3.1 shows the three basic interactive tasks: `enterInformation`, `viewInformation` and `updateInformation`. They create user interface elements to enter, view and update information respectively. As seen in Listing 3.1, these basic tasks include a class constraint on the type of the returned `Task`. This constraint enforces that the type `m` has an instance of the `iTask` type class.

```
enterInformation  :: d [EnterOption m]       → Task m | iTask m
viewInformation   :: d [ViewOption m] m      → Task m | iTask m
updateInformation :: d [UpdateOption m m] m → Task m | iTask m
```

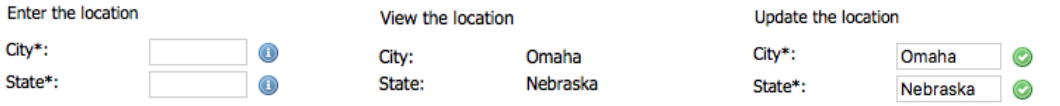
Listing 3.1: iTasks basic interaction functions

Listing 3.2 displays examples of how to use the basic interactive tasks. First, a new ADT called `Location` is introduced. Next, its instance of `iTask` is automatically derived. Following, a new example `Location` is introduced. Finally, new tasks are defined in terms of the basic tasks presented in Listing 3.1.

```
:: Location = { city :: String, state :: String }

derive class iTask Location

location :: Location
location = { city="Omaha", state="Nebraska" }

enterLocation :: Task Location
enterLocation = enterInformation "Enter the location" []

viewLocation :: Task Location
viewLocation = viewInformation "View the location" [] location

updateLocation :: Task Location
updateLocation = updateInformation "Update the location" [] location
```

Listing 3.2: Example of basic iTasks interaction functions

Figure 3.2 displays the user interfaces generated for the basic tasks `enterInformation` (Figure 3.2a), `viewInformation` (Figure 3.2b) and `updateInformation` (Figure 3.2c).



(a) Enter information (b) View information (c) Update information

Figure 3.2: The visual representation of the basic iTasks interaction functions

3.3 Combinators

Although the basic tasks introduced in Section 3.2 allow the user to exchange information with the iTasks application, they are quite limited. In order to allow the user to express more complex behavior, task combinators were introduced. Combinators are functions (usually infix operators) that determine how its argument tasks will be combined into a new task. There are only two fundamental composition combinators: sequence and parallel [20]. All the other combinators in the `iTask` library are derived from these fundamental combinators. Only the derived combinators will be discussed in this document.

3.3.1 Sequential Combinators

Tasks can be executed in sequence using the `>>*` combinator, also called the *step* combinator. Its type signature can be seen on Listing 3.3. The step combinator takes a `Task a` and a list of task continuations as input. A task continuation defines a condition on when to execute another task. It is a predicate that runs on either user actions, task values or thrown exceptions.

```
(>>*) infixl 1 :: (Task a) [TaskCont a (Task b)] → Task b | iTask a & iTask b

:: TaskCont a b
        = OnValue ((TaskValue a) → Maybe b)
        | OnAction Action ((TaskValue a) → Maybe b)
        | E.e: OnException (e → b) & iTask e
        | OnAllExceptions (String → b)

(>>=) infixl 1 :: (m a) (a → m b) → m b   | iTask a & iTask b
(>>|) infixl 1 :: (m a) (m b)     → m b   | iTask a & iTask b
```

Listing 3.3: Sequential combinators

The `>>=` (also called *bind*) is an example of sequential combinator derived from the step combinator. Its first argument is the first task to be executed and its second arguments is a function that takes the first task's value as input and returns a task. The `>>|` combinator is derived from the bind combinator. It executes two tasks sequentially, but the value of the first task is disregarded. On both `>>=` and `>>|`, the second task is executed if either the value of the first task becomes stable, or it is unstable and the user presses a "Continue" button.

3.3.2 Parallel Combinators

Tasks can be executed in parallel using the *parallel* combinator. All the other parallel combinators in the iTasks library are derived from it. Among them, `-&&-` and `-||-` are the most commonly used. The first executes two tasks in parallel and returns a tuple containing the result of both

argument tasks. The `-||-` combinator executes two tasks in parallel, but behaves differently from the `-&&-` combinator. The value of the combined task depends on the stability of the argument tasks. If none of the tasks are stable, the combined task yields the last modified non-stable value. If one of the values becomes stable, the combinator yields it, preserving its stability. The type signatures of both combinators can be seen on Listing 3.4.

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

Listing 3.4: Parallel combinators

3.4 Shared Data Sources

Albeit task combinators are a powerful tool to communicate values among tasks, some applications need to perform ad hoc communication with the external world. Shared Data Sources (SDSs) abstract from implementation details on how resources are accessed. A resource can be, for example, a database, a file, the system time or a shared value in memory.

One approach to create an abstract view on shared data is called a lens [9]. A lens is a bidirectional transformation that provides an interface for reading and writing shared data. Lenses can be extended to allow change notifications: a view is notified when the underlying shared data was modified. Thus, a view always presents updated information. This ability to listen for notifications is called *observing*. Although change notifications are useful, each notification has communication and performance costs. When the shared data is relatively small and seldom updated, no problem arises. Although, if the data is represented by a large data structure that is often updated, broadcasting notifications might impact performance.

This performance impact could be reduced if lenses were able to filter notifications based on which part of the underlying data was modified. Traditional lenses do not provide a mechanism to encode that information. Parametric lenses are an extension to traditional lenses that enable notification filtering according to a *focus* domain [7]. This focus domain is used to encode notification predicates that tell whether a change on the underlying shared source affects the current view. This way, change notifications can be sent only to the views that are affected by the change, reducing unnecessary notifications.

SDSs are implemented using parametric lenses. As it can be seen from Listing 3.5, an `SDS` has three type parameters. The first one is the type of the *focus* domain. The second and third types are the read and write data types respectively. Note that they do not need to be the same type. A `ReadWriteShared` is an SDS where the focus domain is of type `()` (void). A `Shared` is a `ReadWriteShared` where read and write types are the same.

There are four basic functions to operate on SDSs: *get*, *set*, *update* and *watch*. Their type signatures can be seen on Listing 3.5. The `get` operation simply fetches the current value of an SDS. Analogously, the `set` function sets an SDS's value. The `upd` operation sets the SDS's value based on its current value. The `watch` function continuously reads the value of an SDS. The `get`, `set` and `update` operations are atomic: during a reading, no other operation is executed.

```
:: SDS p r w = ...
:: ReadWriteShared r w :== SDS () r w
:: Shared a :== SDS () a a

get   :: (ReadWriteShared a w)          → Task a | iTask a
set   :: a (ReadWriteShared r a)        → Task a | iTask a
upd   :: (r → w) (ReadWriteShared r w)  → Task w | iTask r & iTask w
```

```
watch :: (ReadWriteShared r w)              → Task r | iTask r
```

Listing 3.5: Shared Data Sources definitions

SDSs can be composed to create new SDSs using combinators defined in the iTasks library. In addition, interactive tasks equivalent to the ones described in Section 3.2 are defined for the type `ReadWriteShared`. Their signatures can be seen on Listing 3.6. Both of these interactive tasks *observe* the SDS. Therefore, changes on the `ReadWriteShared` will be automatically displayed.

```
viewSharedInformation ::   d [ViewOption r]      (ReadWriteShared r w) → Task r | iTask r
updateSharedInformation :: d [UpdateOption r w] (ReadWriteShared r w) → Task r | iTask r & iTask w
```

Listing 3.6: SDS interactive tasks

Chapter 4

The mTask EDSL

In some cases, interactions between the iTasks system and the real world could be automated. This is the case for tasks such as reading the room temperature or turning a LED on once a task is completed. Microcontrollers are perfect for this kind of task. They are cheap systems that great for control tasks that involve reading sensors (e.g., temperature, light) and controlling actuators (e.g., motors, LEDs). Due to hardware limitations, microcontrollers can not run iTasks tasks. As an alternative, the mTask EDSL was created. This DSL allows the programming of microcontrollers in Clean using a TOP-like approach [13, 14, 17].

The mTask DSL is a type safe, class-based, multi-view EDSL (Section 2.2.1). It currently has three views: C++ code generation, evaluation and interpretable bytecode generation. An overview of the views is given in Section 4.1.2.

4.1 The Language

Because mTask is a shallowly EDSL, language constructs are represented as functions in the host language. Instead of using functions directly, mTask is composed by type constructor classes, as described in Section 2.2.1. A mTask type constructor class can be seen on Listing 4.1. In this example, the `arith` class is partially presented. An overview of mTask classes will be presented in Section 4.1.1.

```
class arith v where
    lit :: t → v t Expr                                | mTaskType t
    (+.) infixl 6 :: (v t p) (v t q) → v t Expr        | type, +, t & isExpr p & isExpr q
```

Listing 4.1: A mTask class

Language constructs are of the form `v t p` where `v` is the view, `t` is the type of the construct and `p` is the construct's role. The `lit` function lifts a value to the mTask domain. The `+.` infix operator adds two expressions (represented by the `isExpr` class constraint) and returns another expression. The constraints `type` and `mTaskType` ensure that only mTask types can be used. The constructor role specifies whether the constructor is an updatable, an expression or a statement [14, 17]. The definitions of the constructor roles and the `isExpr` class along with its instances can be seen in Listing 4.2.

```
:: Upd  = Upd
:: Expr = Expr
:: Stmt = Stmt
```

```
class isExpr a :: a
instance isExpr Upd
instance isExpr Expr
```

Listing 4.2: mTask construction roles

Views are instances of mTask classes. Due to the multi-class nature of mTask, a view can choose which language constructs it supports by selecting which classes it implements. Moreover, new language constructs can be added to the language without the need to change existing code.

4.1.1 Overview of the Classes

The mTask EDSL is formed by many type constructor classes. Here, only the classes that are most relevant to the research are presented. Some class constraints were omitted to ease understanding.

Expression: There are two classes to create expressions in mTask: `arith` and `boolExpr`. They model constructs for arithmetic and boolean expressions respectively. The `arith` class contains operators for addition, multiplication, subtraction and division in addition to a function to lift values to the mTask domain. The `boolExpr` contains operators over booleans (e.g., conjunction and disjunction) in addition to equality and inequality operators. Shortened versions of these classes can be seen in Listing 4.3.

```
class arith v where
        lit :: t → v t Expr
        (+.) infixl 6 :: (v t p) (v t q) → v t Expr | + t
        ...

class boolExpr v where
        (&.) infixr 3 :: (v Bool p) (v Bool q) → v Bool Expr
        Not           :: (v Bool p)            → v Bool Expr
        (==.) infix 4 :: (v a p) (v a q)       → v Bool Expr | == a
        (<.)  infix 4 :: (v a p) (v a q)       → v Bool Expr | < a
        ...
```

Listing 4.3: mTask expression classes

Control flow: The `IF` class implements *if* constructs. The `IF` function implements a *if-then-else* statement and the `?` infix operator implements an *if-then* statement. Given that mTask tasks can be executed periodically, loop constructs are unnecessary and are not part of mTask. The `seq` class contains the the monadic bind operator (`>>=.`) for mTask. In addition, it contains a variant of the monadic operator where the result of its first argument is disregarded. This operator is equivalent to the semicolon in imperative languages. The `retrn` class contains a single function that terminates the task. Control flow classes can be seen in Listing 4.4

```
class IF v where
  IF :: (v Bool p) (v t q) (v s r)  → v () Stmt | isExpr p
  (?) infix 1 :: (v Bool p) (v t q) → v () Stmt | isExpr p
class seq v where
  (>>=.) infixr 0 :: (v t p) ((v t Expr) → (v u q)) → (v u Stmt)
```

```
  (:.) infixr 0 :: (v t p) (v u q) → v u Stmt
class retrn v where
  retrn :: v () Expr
```

Listing 4.4: mTask control flow classes

Shared Data Sources The `sds` class contains functions to create Shared Data Sources. The `sds` function is used to create updatable SDSs and the `con` function is used to create constant SDSs. Both use the technique described in Section 2.2.1 to guarantee a type-safe usage of SDSs. The `sdspub` class contains a construct to publish SDSs. The `assign` class contains a single function to enable assignment in mTask. SDS classes can be seen in Listing 4.5. Notice that the functions below enforce the construct role `Upd` to ensure that only updatables are used.

```
:: In a b = In infix 0 a b
:: Main a = {main :: a}

class sds v where
  sds :: ((v t Upd)  → In t (Main (v c s))) → (Main (v c s))
  con :: ((v t Expr) → In t (Main (v c s))) → (Main (v c s))
class sdspub v where
  pub :: (v t Upd) → v t Expr
class assign v where
  (=.) infixr 2 :: (v t Upd) (v t p) → v t Expr | isExpr p
```

Listing 4.5: mTask SDS classes

Input and output There are constructs to handle both analog and digital input and output. The classes `dIO` and `aIO` contain functions to handle digital and analog I/O, respectively. Both classes create updatables that can be used to read from and write to a pin. The `userLed` class contains functions to turn LEDs on an off. Input/output classes can be seen in Listing 4.6.

```
:: DigitalPin = D0 | D1 | D2 | D3 | D4 | D5 |D6 | D7 | D8 | D9 | D10 | D11 | D12 | D13
:: AnalogPin  = A0 | A1 | A2 | A3 | A4 | A5
:: UserLED    = LED1 | LED2 | LED3

class dIO v where
       dIO :: DigitalPin → v Bool Upd
class aIO v where
       aIO :: AnalogPin  → v Int Upd
class userLed v where
  ledOn  :: (v UserLED q) → (v () Stmt)
  ledOff :: (v UserLED q) → (v () Stmt)
```

Listing 4.6: mTask I/O classes

4.1.2 Overview of the Views

Currently, mTask has three views: C++ code generation, evaluation and bytecode generation.

The C++ code generation view translates language constructs to Arduino's dialect of C++. The Arduino IDE compiles the C++ source code to machine code for the microcontrollers. It

is convenient to generate C++ code instead of machine code because it saves us from the task of generating code for different microcontrollers. In addition, C++ gives us the level of control necessary to handle low level input/output operations. This view consists of a function that modifies a compilation state, `CODE`. The state is a record that stores the generated code along with some information to generate identifiers and to keep track of source code indentation. At the end of compilation, the `CODE` record is transformed into C++ code which can be saved into disk, loaded into the Arduino IDE and uploaded to microcontrollers.

The evaluation view translates language constructs into Clean programs. It consists of a function that modifies an evaluation state. The state is a record that stores tasks, program variables and input/output information. Given that programs running on microcontrollers are hard to debug, one can benefit from this view to find program errors. This view can be used to build a mTask simulator using iTasks where the user can observe the state of the program on each loop of the microcontroller.

The last mTask view transforms language constructs into interpretable bytecode. Since the research focused on this view of mTask, it will be discussed in more detail in Section 4.2.

4.2 Interpreted mTask

4.2.1 Motivation

Although the C++ code generation view works as expected, it poses a limitation. Tasks generated by this view are static: once they are compiled and uploaded, they cannot be changed. If the user wishes to change the current task or add new tasks to the microcontroller, the program has to be recompiled and reuploaded. This presents two problems. First, microcontrollers have a limited amount of write cycles in their program memory [17]. Therefore, repeated uploading of new programs is not desired. Second, due to the nature of microcontrollers and the IoT, such devices are often located on places that are hard to reach. Often, microcontrollers must be physically reached and plugged to a computer in order to be reprogrammed. Ideally, tasks would be uploaded without the need to plug the devices into a computer. Although some microcontrollers may be remotely reprogrammed using Over-the-Air programming, this technique erases the device's RAM. Therefore, the tasks that were executing before the device reprogramming are lost. In an ideal scenario, sending a new task to a device would preserve the current tasks being executed.

To overcome that limitation, a new view of mTask was created [17]. This view generates interpretable bytecode rather than C++ code. The bytecode can be interpreted by a runtime system in the device (the *client*). This runtime system (the *engine*), is written in C and can be compiled and uploaded using the Arduino IDE. Tasks and SDSs are sent to the client dynamically. Therefore, devices are programmed once but can execute tasks dynamically. Additionally, this setting can be more robust than the static mTask. If the communication with a device fails, the server can dynamically send the tasks that were running on it to another suitable device.

4.2.2 Communication Protocol

Devices can be connected either via Serial communication or TCP. The server and the client communicate via a protocol based on messages. Outgoing messages are of type `MTaskMSGSend` — messages are always named from the server's perspective. There are messages for task addition and deletion, shutdown request, SDS addition and update and specification request. The definition of the `MTaskMSGSend` ADT can be seen in Listing 4.7.

```
:: BCValue = E.e: BCValue e & mTaskType e

:: MTaskMSGSend =
      MTTask MTaskInterval String
    | MTTaskDel Int
    | MTShutdown
    | MTSds Int BCValue
    | MTUpd Int BCValue
    | MTSpec
```

Listing 4.7: Communication protocol: sent messages

The client communicates with the server via messages of type `MTaskMSGRecv`. There are messages for task acknowledgment and deletion, SDS acknowledgment, deletion and publication, debugging messages, device specification and empty messages. The definition of the `MTaskMSGRecv` ADT can be seen in Listing 4.8. The `MTaskDeviceSpec` data type contains the specification of the device, i.e. its stack size, memory size, number of digital and analog pins.

```
:: MTaskMSGRecv =
      MTTaskAck Int Int
    | MTTaskDelAck Int
    | MTSDSAck Int
    | MTSDSDelAck Int
    | MTPub Int BCValue
    | MTMessage String
    | MTDevSpec MTaskDeviceSpec
    | MTEmpty
```

Listing 4.8: Communication protocol: received messages

4.2.3 The Client

The client runs a loop function that runs repeatedly until a *shutdown* message is received. The loop consists of two pieces: checking for incoming messages and running the task scheduler. The first step is straightforward: it checks the input buffer and processes any messages that might be in it. The task scheduler runs tasks based on task intervals. Tasks can run once (`OneShot`), repeatedly based on an interval (`OnInterval`) or based on an interruption (`OnInterrupt`). The interpreter is responsible for the execution of a task's bytecode.

4.2.4 The Simulator

During my Research Internship, I developed an iTask simulator for the interpreted mTask. The motivation behind it is the same as the simulator for the static mTask: programs running in microprocessors are hard to debug. The simulator mimics the C engine in many aspects, including communication, task scheduling and instruction interpretation.

The simulator provides a web interface where the user can inspect the communication channels and the simulator state. The simulator state contains data about the simulator clock, memory, stack, tasks, SDSs and peripherals (pins, LED, etc). The web interface allows peripheral values to be set manually. This is a great addition to the mTask development environment, given that sensor values can not be easily simulated on microcontrollers. In addition, users can set breakpoints on bytecode instructions and inspect the state of the simulator on specific points.

The simulator offers two modes: manual and automatic. Manual mode requires user interaction via the web interface to run. Manual simulation can be performed via either big or small steps. These two step options offer different levels of abstraction to the user. Big steps execute one entire engine loop at a time. Small steps execute each bytecode instruction individually. Automatic mode executes the simulator without the need of user interaction. It is particularly useful when the programmer needs to simulate a device but does not want to step manually through its execution. Many simulators on different modes can run simultaneously.

4.2.5 Devices

The mTask library provides functions to interact with devices. Their type signatures can be seen in Listing 4.9. Some of these functions take an `MTaskDevice` as input.

```
:: Channels :== ([MTaskMSGRecv], [MTaskMSGSend], Bool)

:: MTaskDevice

class channelSync a :: a (Shared Channels) → Task ()

withDevice :: a (MTaskDevice → Task b) → Task b | channelSync a

liftmTask :: MTaskDevice MTaskInterval (Main (ByteCode a Stmt)) → Task ()
```

Listing 4.9: Device interaction functions

A server communicates with devices by sending and receiving messages (Section 4.2.2). Message communication is performed by synchronizing a device's communication channels with the actual device. Device channels (represented by the `Channels` data type) contain messages received from the device (`MTaskMSGRecv`), messages to be sent to the device (`MTaskMSGSend`) and the channels' status (`Bool`).

The `channelSync` class is responsible for device communication and is implemented by every mTask device type (i.e. TCP, Serial, simulator). It contains only one function — also called `channelSync` — which synchronizes a device's channels. The `withDevice` function takes a synchronizable device specification (any type with an instance of the `channelSync` class) and a function as input. This argument function is the actual Task to be performed with the device. The `withDevice` task performs three tasks in parallel: it synchronizes the device channels (using `channelSync`), processes incoming `MTaskMSGRecv` messages and executes its argument task.

The `liftmTask` function sends an mTask task to the given device. Its first argument is the device to which the task will be sent. Its second arguments is the task interval. Its last argument is the actual mTask task to be sent. The `liftmTask` task stabilizes whenever its argument task finishes executing on the device.

4.3 Examples

Two examples of mTask tasks can be seen in Listing 4.10. The first example, `switch`, turns an LED on and off based on a switch connected to digital pin `D0`. The second example, `curtains`, opens and closes curtains based on the room lighting. The curtains' controller is connected to digital pin `D0` and the light sensor is connected to analog pin `A0`. When the light sensor value is greater then 3, the curtains open. Otherwise, the curtains close. In addition, an alarm (represented by the `alarm` SDS) is triggered when the curtains open. Once the curtains open and the alarm is triggered, the task terminates.

```
switch :: Main (v () Stmt)
switch = { main =
        IF (dIO D0) (
                ledOn (lit LED1)
        ) (
                ledOff (lit LED1)
        )}

curtains :: Main (v () Stmt)
curtains = sds λalarm = False In { main =
        IF (aIO A0 >. lit 3) (
                dIO D0 =. lit True :.
                alarm =. lit True :.
                pub alarm :.
                retrn
        ) (
                dIO D0 =. (lit False)
        )}
```

Listing 4.10: Examples of mTask tasks

Chapter 5

The Application

The first step to answer the research question was to choose an application to develop. This chapter presents the criteria used in the selection process and the application chosen: home automation.

5.1 Selection Criteria

The application developed during the research should ideally display characteristics inherent to IoT applications [21, 18, 22]. Although, some of there characteristics — energy consumption, performance and security — were not considered during the research. These aspects were ignored because the development of iTasks did not take them into consideration. Additionally, some criteria were added based on the research question.

The following criteria were used to choose an application:

Suitable The application should solve a problem that is suitable for mTask. This narrows the choice to IoT applications that can be developed for platforms supported by mTask.

Non-trivial Trivial applications (e.g. an LED blinking) were previously developed [17]. Therefore, the application should not solve a trivial problem — e.g. a simple hallway motion-activated light sensor. It should go beyond a purely reactive system. It does not have to solve a novel problem, but its development should be adequately challenging.

Simple Due to time constraints, it should be simple enough to be developed during this research. Since building a full-pledged application is not the goal of the research, some concerns as feature completeness, user experience design and security are not taken into account. Its source code should not be too complex.

Interesting It is not enough that the application is suitable and technically good. It should tackle an existing, interesting problem. Ideally, a problem which users inserted into the application domain would be willing to pay for a solution.

Significant The application should somehow improve the environment it is inserted into. Examples are accelerating an assembly line, saving commute time, improving one's health or well being or reducing operational cost.

Comprehensible Its domain and main features should be easily understandable by non-domain experts. Its functionality details and operational features might require specific knowledge, but the application should be easily described on a high level to someone who is not familiar with its domain. Comprehensibility is relevant because it improves the application and therefore the research's reachability.

Robust The application should be able to handle errors to some extent. It should at least be able to detect and communicate device disconnection. Ideally, it would automatically migrate tasks from disconnected devices to available devices whenever possible.

Highly connected It should support multiple devices simultaneously. These devices should be able to exchange information (e.g. sensor values) when suitable. Ideally, the devices would be connected wirelessly.

Dynamic The application should not be static. Given that the interpreted version of mTask (Section 4.2) is being explored, its dynamic nature should be exploited. The application domain should naturally allow dynamicity. Ideally, tasks would be sent to and removed from devices regularly.

Diverse It should use as many peripherals as possible. Since IoT applications often handle a heterogeneous group of peripherals, it is important that a diverse group of sensors and actuators is used. The application should not restrict itself to a couple of peripherals.

Extensive The application should use mTask features extensively. Given that it is testing mTask's capabilities, it is important that the application tests as many mTask features as possible. There is a correlation between the number of features used by the application and the certainty about mTask's abilities.

5.2 Home Automation

Many potential IoT applications were considered (e.g. greenhouse automation) to be developed during research. After a systematic selection process based on the selection criteria described on Section 5.1, a home automation solution was chosen. A detailed analysis of why this domain was chosen is presented in Section 5.4.

Home automation might refer to different levels of automation of home tasks. By definition, any tool or machine that automates a home task constitutes a home automation solution. Historically, home automation became popular with the advent of distributed electricity. Daily tasks as dishwashing or drying clothes were automated by appliances that today are common in many households around the world.

In the last decades, home automation gained another meaning with the invention of electronic solutions that control virtually any electronic in a house. Lighting, air conditioners, heaters, entertainment systems and doors are common components controlled by home automation solutions. Frequently, these systems are composed by a central control unit with a user interface (e.g. computer, tablet, smartphone, wall control panel), a communication channel (e.g. Bluetooth, LAN, Internet, infrared) and devices to be controlled (e.g. lamp, air conditioning unit, doors, TV, appliances).

Automated tasks might be as simple as turning a light on when someone enters the room, controlling the heater based on a target room temperature, locking the main door at a set time

and closing the curtains based on the amount of natural light outside. They might also be more elaborated as automatically turning the coffee maker on at 8:00 AM on work days, but only if somebody is at home.

5.3 Application Description

The home automation application developed is called Autohouse. It enables and manages the automation of a home (hereafter referred to as *smarthome*) using mTask. A smarthome is composed by rooms that can be added and removed by its user. Each room contain devices (called *units* in the application) that execute tasks chosen by the user.

A smarthome is managed via the web control panel. Multiple instances of the control panel can run simultaneously. There, the user has access to the main features of the application:

- Manage home: add and remove rooms to the smarthome.
- Manage room: add and remove units to a room.
- Send task: send a new task to one of the available units.
- Stop task: delete a task that is currently running on a unit.
- Inspect unit: see which tasks are running on a unit.

5.3.1 Architecture

Autohouse is a centralized solution: a server (ideally located in the home) is the central communication hub for both users and devices. User communication is accomplished via the web control panel, which is hosted in the server. Device communication is accomplished via the mTask library and its communication protocol described in Section 4.2.2. Figure 5.1 displays an example architecture of Autohouse deployed on a home with three units.

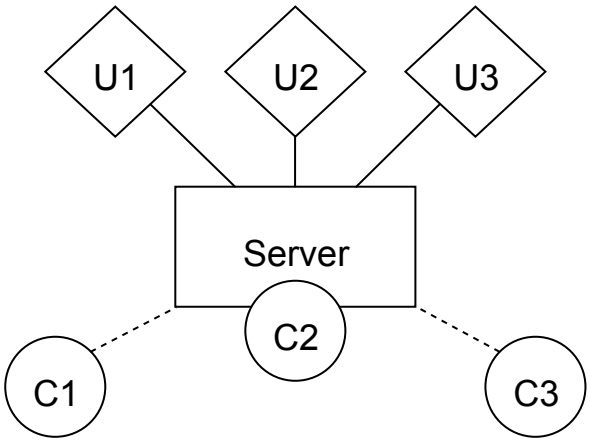


Figure 5.1: Autohouse Architecture

In Figure 5.1, units are represented as diamonds, the server is represented as a rectangle and clients accessing the web control panel are represented as circles. Rooms are just an abstraction

layer to ease the use of Autohouse and therefore are not represented in its architecture. Units are microcontrollers equipped with peripherals (sensors, actuators) and with a communication interface. The server can be any computer with networking capabilities that is supported by Clean and that has enough resources to run the application. A client is a device accessing the control panel using a web browser. Note that the server itself may be a client (C2 in Figure 5.1), since it can access the control panel via a web browser. In addition, if the server is exposed to the Internet, the control panel can be accessed remotely. It is important to point that the development did not regard security. When exposed to the Internet, the application can be accessed by anyone.

Communication between the server and the units must be persistent (represented by the continuous line between them in Figure 5.1). If the communication is interrupted, the server interprets it as a disconnection. Communication between the server and remote clients can be transient (represented by the dotted line between them in Figure 5.1). Since remote clients are just managing the application, their connection does not have to be persistent. The server can access the control panel locally and therefore does not communicate wirelessly (represented as C2 attached to the server in Figure 5.1).

Following the "highly connected" criterion introduced in Section 5.1, units should connect to the servers wirelessly. The mTask library supports two types of connections: Serial and TCP. Although, which technology is used to establish these connections is irrelevant to the end user. Therefore, Autohouse is technology agnostic in regard to communication between units and the server. Units can, for example, establish a serial connection via Bluetooth or a TCP connection via WiFi. As a consequence, Autohouse does not enforce a wireless connection between the units and the server. The user decides which communication technology is more adequate to their needs.

5.3.2 Tasks

The main purpose of Autohouse is to send automation tasks to units so they can be executed without human supervision. Therefore, the tasks the application supports play a key role in Autohouse. The application has a list of predefined tasks that are relevant to the house automation domain. The user can pick tasks from this default task list and select a unit to send it to. The standard task list might be extended by adding new tasks to the `Programs` module. The Autohouse standard tasks are listed below.

- Control a light based on a switch.
- Turn a light on when movement is detected.
- Lock a door when it gets dark.
- Open the garage door if a car is reversing.
- Turn a fan on when it gets humid enough, but turn it off when something is about to hit it.
- Open and close windows based on a target temperature range.
- Open and close curtains based on the amount of light in the room.
- Manage the air conditioning unit and the heater based on a target room temperature and on the actual room temperature.

5.3.3 Devices

Three platforms are supported by mTask: Arduino, POSIX and Mbed. Arduino [2] was chosen as the main platform for Autohouse. Arduino is an open-source electronics prototyping platform which has single-board microcontrollers specifications and software to support them. It was chosen because of its open-source nature, popularity, well-established community and open-source code availability.

The Arduino platform currently has many boards with different specifications and purposes. The Arduino Uno Rev3[1] was chosen as the target device because of its popularity, extensibility (using shields[2]), cost and limited resources. The Arduino Uno is based on the ATmega328P microprocessor, which has only 2 KB of RAM. This memory limitation is a desired characteristic, given that mTask's suitability for microcontrollers is being tested and that such devices might have extremely limited resources. The Arduino Uno operates at a frequency of 16 MHz, has 6 analog pins, 14 digital pins, 32 KB of flash memory and a built-in LED. Its digital and analog pins are often used to interface with peripherals, including sensors and actuators.

Sensors and actuators are extremely relevant to Autohouse because they allow the application to interface with the real world without human interference. Examples of sensors are temperature, humidity and movement sensors. Examples of actuators are buttons, switches, LEDs and motors.

5.3.4 Sensors

Autohouse relies on the interpreted mTask and its Arduino client for peripheral support. Therefore, it supports the same sensors the Arduino client does. Digital and analog pins can be controlled explicitly, but mTask also provides native support for some sensors. Namely, for the DHT22 temperature and humidity sensor, the HCSR04 ultrasonic sensor, digital light sensor, Grove analog light sensor (P) V1.1 and Passive Infrared Sensor (PIR). Native sensor support allows programming in a high level of abstraction, in which sensor values represent their semantics. For example, reading from a temperature sensor returns a `Temperature`, an ADT that represents temperature in Celcius. Some of these sensors were not previously supported by mTask. Section 6.2.3 describes how these peripherals were incorporated into mTask.

5.3.5 Actuators

Similarly to sensors, Autohouse relies on the interpreted mTask and its Arduino client for actuator support. Some actuators (e.g. switches, buttons) can be directly controlled through pins and therefore do not require custom support. Other actuators, though, require custom support to be practically used. The interpreted mTask provides native support for LCD displays (to print integers only), LEDs and servos. Servos were not previously supported by mTask. Section 6.2.3 describes how servos were incorporated into mTask.

5.4 Application Analysis

The selection process that chose Autohouse as the research application was guided by the selection criteria presented in Section 5.1. In this section, the Autohouse choice is motivated by analyzing the application under those same criteria.

[1] Arduino Uno Rev3. Available at `https://store.arduino.cc/arduino-uno-rev3`. Accessed on August 26th 2018.

[2] Arduino Shields. Available at `https://www.arduino.cc/en/Main/arduinoShields`. Accessed on August 26th 2018.

Suitable Autohouse is suitable for mTask. Its components are small devices connected to a server that (if connected to the internet) can be accessed anywhere. In addition, Autohouse targets Arduino Uno, a board supported by mTask.

Non-trivial It is not a trivial application. The complexity of Autohouse goes beyond a simple reactive system. It is an application with a user interface where its user can dynamically add devices and send automation tasks to such devices. Its tasks can be simple reactive tasks but can also contain complex logic based on input from different sensors and devices.

Simple Autohouse was simple enough to be developed during the research. Thanks to the prototyping nature of iTasks, its development could abstract from many technical details and focus on design decisions.

Interesting Home automation is becoming increasingly popular and its industry is growing consistently[3]. There is no industry standard and the market is highly fragmented. Therefore, Autohouse solves an existing, interesting problem.

Significant Autohouse might improve many aspects of its user's life. For example, Autohouse tasks can save time (e.g. automatically opening the garage door when a car moves backwards). They can also save energy (e.g. turn the hallway light off if no one is walking by). They might also improve the user's well being (e.g. automatically regulate room temperature). In addition, tasks can be simply convenient (e.g. open the curtains when it is bright outside).

Comprehensible The home automation domain is comprehensible to most people. We all live in homes and can relate to most of the tasks Autohouse offers. Therefore, no explanation of the application domain is required to comprehend it.

Robust Autohouse is a robust application when it comes to device disconnection. If a unit is disconnected from the server, the system migrates the tasks that were running on that unit to another, suitable unit. The application does not support server fault tolerance. Although the distributed version of iTasks could have been used, Autohouse runs on the single server version instead [19].

Highly connected Autohouse supports simultaneous devices. Although wireless connections are supported (e.g. Bluetooth), Autohouse does not enforce them (Section 5.3).

Dynamic The Autohouse system allows for dynamic addition and removal of devices. In addition, a user might send tasks to devices dynamically. Home automation is a domain that naturally requires dynamicity. It is expected that Autohouse users send and remove tasks from devices on a daily basis.

Diverse Autohouse supports five different sensors and three actuators. Therefore, in total, eight peripherals are integrated into the application. Although this number is small when compared to some commercial applications out there, it is enough to display Autohouse's diversity.

[3]Global Home Automation Market Growth Opportunities 2017-2022 - Market is expected to reach an estimated $75.2 billion. Available at `https://markets.businessinsider.com/news/stocks/global-home-automation-market-growth-opportunities-2017-2022-market-is-expected-to-reach-an-estimated-75-2-billion-1007431226`. Accessed on August 27th 2018.

Extensive The application tests mTask features thoroughly. Regarding the mTask language, Autohouse's default tasks use all of the language constructs implemented by the mTask's interpreted view (Section 4.2). In addition, it makes use of mTask functionality to connect to devices and to send tasks to it. Autohouse benefits from mTask abstraction layer over devices and does not handle them directly.

Chapter 6

Application Development

After the application domain was chosen, development began. Although Autohouse was developed during the research, it was not the research object. It was used merely as a tool to assess mTask's capabilities and thus answer the research question. Therefore, this chapter will not focus on Autohouse itself, but on the limitations of mTask unearthed during its development. A quick overview of its development is given in Section 6.1. Some limitations of mTask were overcome and are described in Section 6.2. Other limitations remain and are described in Section 6.3. Finally, Section 6.4 describes how automatic task migration was accomplished without modifying mTask.

6.1 Development Overview

Autohouse is an application developed using Clean, iTasks and mTask. Due to time constraints, it was thoroughly tested only on macOS 10.13 (High Sierra). It was tested on Linux (Ubuntu 16.10) on early stages of development. Autohouse's source code is available at its GitHub repository[1].

6.1.1 Application Architecture

Autohouse development started just like many iTasks applications: defining ADTs and SDSs. The application has ADTs that model key concepts in the home automation domain: `House` (representing the smarthome), `Room` and `Unit` (representing a device). A `House` is a list of `Room`s and a `Room` is essentially a list of `Unit`s.

All the application data lives in one SDS that represents the entire house. Other SDSs (e.g. for rooms and units) are derived from this main SDS using parametric lenses [7]. Rooms and units have unique ids that are used to locate them in the house SDS. The default automation tasks are stored in a list of `Program` — in Autohouse's source code, automation tasks are named *programs* to avoid name clashing with iTasks' `Task` ADT. The `Program` record contains all the information inherent to an automation task.

The application contains three main tasks running in parallel: manage house, manage units and send new task. The first one lets the user create and edit rooms. The second allows the user to inspect, send tasks to and disconnect units. The last task lets the user pick a mTask task to send to a device that is compatible with it.

Once the iTasks foundation was created, the mTask development could begin.

[1] Autohouse on GitHub. Available at `https://github.com/matheusamazonas/autohouse`.

6.1.2 Using the Simulator

As expected, device features were first implemented using simulators (Section 4.2.4). These devices proved to be great for early stages of development. First, multiple simulators can be easily instantiated, which was particularly useful when physical devices were not available yet. Simulators allowed the development of peripheral tasks even before some peripherals were available for testing. Additionally, given that simulators are highly customizable, some features that depended on different device configurations were tested. Such freedom to quickly choose between different device configurations does not exist when dealing with physical devices.

Although the simulator was particularly helpful during early stages of development, it proved to be a great debug tool during later stages as well. Whenever a device behaved unexpectedly, the same environment was reproduced using a simulator. Then, using the debugging UI, the device's state (i.e. memory, tasks, program counter, peripheral values) could be inspected. Using the simulator as a debug tool became standard practice during Autohouse's development.

6.1.3 Device Communication

Once prototyping using the simulator was over, development moved to physical devices. But one problem had to be solved before deploying mTask tasks on Arduinos: wireless device communication. Ideally, Autohouse would communicate with its devices wirelessly. But so far, only Serial connections over USB were used to connect to Arduinos. Two wireless options were taken into consideration: Wi-Fi (through TCP) and Bluetooth (through Serial).

The first option tested was Wi-Fi, specifically with the ESP8266. The ESP8266 is a system on a chip with a microcontroller, a full TCP/IP stack and Wi-Fi. It can be used as a Wi-Fi module (giving other microcontrollers Wi-Fi capabilities) and as a standalone microcontroller. In Autohouse's setting, it would be used to give the Arduino boards Wi-Fi capabilities. A microcontroller can use Hayes commands[2] to control the ESP8266 as a Wi-Fi module. Thus, a library was required to interface with it. A couple of open-source libraries were tested[3,4], but they either did not offer some necessary features (e.g. retrieve the list of connected clients) or behaved unexpectedly (e.g. losing received messages). After some unsuccessful attempts to adapt the existing libraries, Wi-Fi was put aside and Bluetooth was considered as an alternative.

The Bluetooth module tested was the HC-05, a Bluetooth 2.0 Serial Port Protocol (SPP) module. Since this module runs through Serial, it can be directly connected to the Arduino board's Serial pins (`TX` and `RX`). Incoming data is preprocessed by the HC-05 and send directly to the device's Serial port. Therefore, no client data processing is required to transmit data via Bluetooth. Additionally, no code changes were necessary to support Bluetooth connection. When compared to Wi-Fi, Bluetooth brought clear advantages — i.e. it works out of the box and does not required additional code — and therefore was chosen to be used as Autohouse's wireless solution.

6.1.4 Device Deployment

Once the wireless communication was set up, development moved to physical devices. First, all peripherals were tested using a single Arduino board to ensure that they were compatible. Once

[2]Hayes command set - Wikipedia. Available at `https://en.wikipedia.org/wiki/Hayes_command_set`. Accessed on September 14th 2018.

[3]WiFiEsp on GitHub. Available at `https://github.com/bportaluri/WiFiEsp/tree/master/src`. Accessed on September 14th 2018.

[4]ESP8266wifi on GitHub. Available at `https://github.com/ekstrand/ESP8266wifi`. Accessed on September 14th 2018.

all peripherals were tested using mTask tasks, more devices were added to the Autohouse system.

The test system contained five Arduino Uno compatible boards. Each board was equipped with a HC-05 Bluetooth module, two LEDs, two push down buttons, a PIR and temperature, humidity, ultrasonic and analog light sensors. The boards had similar capabilities in order to test task migration (tasks should only migrate to devices that are compatible with it). Only one board was equipped with a servo. This choice was also motivated by the task migration feature: if a device running a task that uses a servo disconnects from the server, its task should not migrate because no other device has a servo.

Once device deployment finished, Autohouse's standard task list was created and tested. Autohouse's version[5] 0.1.0 corresponds to the end of this development phase.

6.2 Changes to mTask

Limitations of mTask surfaced during the development of Autohouse. Some of these limitations were overcome by changing mTask. These changes are described below.

6.2.1 Variables

Since mTask is an imperative language, it would benefit from mutable data features. Although there are no mTask constructs to represent variables, SDSs might be used as updatable data containers. In such a setting, an SDS is created for each desired variable. This trick brings updatable data storage to mTask, but it prompts two problems.

First, there is no separation of concerns. Variables and SDSs should be, by definition, different things. A variable is a *local* updatable data storage in memory. An SDS is an abstraction layer over any kind of shared data, including data in memory. Using an SDS locally goes against what a *shared* data source represents. Second, when SDSs are sent to devices, they are not attached to a specific task. Also, on the current version of mTask, there is no way to establish whether an SDS belongs to a given task. As a consequence, SDSs are never deleted from devices. Variables, on the other hand, are always bond to a specific task and could be removed with their correspondent task altogether, saving space in the device's memory. Thus, mTask could benefit from a language construct for variables.

The `vari` class was created to fill this gap. It contains two functions: `vari` and `con`, representing variable and constant data storage respectively. Its definition can be seen in Listing 6.1. From a language construct point of view, the `sds` and `vari` classes do not differ much. Both classes contain constructs that might be used as updatables and as expressions. But there are two differences between these classes. First, `vari` contains a construct for constant data: `con`. Second, `vari` functions expect a value of type `t` as its initial value (seen as the first argument of `In` in Listing 6.1). The `sds` function expects a `(Shared t)` instead. The biggest difference between the `sds` and `vari` classes regards their behavior on the interpreted view of mTask. Variables belong to a task and will live as long as the task lives. SDSs are not bound to a task and will live in the device indefinitely.

```
:: Vari  = Vari

instance isExpr Vari
instance isUpd Vari
```

[5] Autohouse release 0.1.0 on GitHub. Available at `https://github.com/matheusamazonas/autohouse/releases/tag/0.1.0`.

```
class vari v where
  vari :: ((v t Vari) → In t (Main (v c s))) → (Main (v c s))
  con ::  ((v t Expr) → In t (Main (v c s))) → (Main (v c s))
```

Listing 6.1: The `vari` class

Listing 6.2 displays an example of variables in mTask: the task `blink`. This task blinks `LED1` based on the value of variable `v`. The variable `v` is created using the `vari` construct. Its value is updated using the `=.` infix operator, similarly to SDSs. It can also be used as a boolean expression, as the condition to an `IF` construct.

All Autohouse automation tasks must be of type `Main (v () Stmt)`. The `noOp` after the attribution in Listing 6.2 is required to ensure that the program matches that type. The `noOp` construct is a wild card used whenever the type of a construct does not match its desired type.

```
blink :: Main (v () Stmt) | program v
blink = vari λv = False In { main =
        IF (v) (
                ledOn (lit LED1)
        ) (
                ledOff (lit LED1)
        ) :.
        v =. Not v :. noOp
        }

class noOp v where noOp :: v t p
```

Listing 6.2: Example of the usage of variables in mTask

The addition of variables to the language required changes on mTask's communication protocol (Section 4.2.2). When a task is sent to a device, its variables must be sent as well. Therefore, a `MTTask` message must include the variables used by the given task. Variables are modelled in the `BCVariable` record. A variable contains a unique (within a task) identifier and its initial value. The `BCVariable` record and the communication protocol change can be seen in Listing 6.3.

```
:: BCVariable = { vid :: Int, vval :: BCValue }

:: MTaskMSGSend
        = MTTask Int MTaskInterval [BCVariable] String
         ...
```

Listing 6.3: Change in mTask's communication protocol to accommodate task variables

Additionally, the simulator and the client engine were modified to support task variables. When a task is received, its variables are stored. During task execution, variables are fetched and assigned similarly to SDSs. When a task terminates, its variables are removed from the device.

6.2.2 Peripheral Code

The mTask library already supported some of the peripherals Autohouse planned to use: LEDs, analog and digital pins. Although, new peripherals (e.g. light, temperature and humidity sensors) were required by some of the default automation tasks. Following the natural development process of an mTask application, these peripherals were first emulated using the simulator.

As more peripherals were implemented, it was clear that the workflow required to add a new peripheral to the system could be improved.

Adding a new peripheral required changes on different parts of mTask. An overview of the necessary changes can be seen below.

- A new class that represents the peripheral is added to the language.
- Depending on the peripheral, a new ADT is created to represent its values (e.g. `DigitalPin`).
- New bytecode instructions are created.
- Bytecode encodings are updated to support the new instruction and the possibly new ADT.
- The `MTaskDeviceSpec` record is modified to include a flag for the new peripheral.
- The simulator interpreter is updated to handle new bytecode instructions.
- The C client is modified to handle the new peripheral.

The changes on the C client code depended heavily on the type of peripheral being implemented. Changes on the Clean code though, were often similar. Previously, peripheral code was scattered around the mTask library. Peripheral classes were inside the `Language` module along with possibly new ADTs. Instances of the peripheral classes for each mTask view were in the respective view's module. The simulator interpreter contained peripheral-specific code. Bytecode encodings for basic types were mixed with encodings for peripheral data types. Overall, adding a new peripheral was particularly cumbersome and extremely error-prone. Finally, there was no separation of concerns whatsoever.

A new modular code architecture for peripherals was introduced to solve the problems described above. It aims to remove peripheral-specific code from mTask core and simulator modules. In this architecture, each peripheral should be defined in its own module. Its type class, ADTs, bytecode encodings and view instances are defined in that same module. The simulator does not have any peripheral-specific code. Instead of explicit fields for each peripheral, the simulator state record (`SimState`) contains a list of `Peripheral`. This new data type is a wrapper around every mTask peripheral. Its definition can be seen in Listing 6.4.

```
:: Peripheral = E.e: Peripheral e & peripheral e

class peripheral e | iTask e where
        processInst :: BC e → State SimState (e,Bool)
```

Listing 6.4: The `Peripheral` class

The `peripheral` class was created to enable the removal of peripheral-specific code from the simulator interpreter. Its only function, `processInst` defines how a peripheral should interpret bytecode instructions (`BC`). Naturally, a peripheral should only interpret instructions that are relevant to it. The simulator interpreter executes one instruction at a time. If an instruction belongs to mTask's core instruction set (excluding peripheral instructions), the interpreter executes it immediately. If the instruction does not belong to the core instruction set, it is assumed to be a peripheral instruction and it is presented to all simulator peripherals using the `processInst` function. Once a peripheral responds to an instruction (represented by the `Bool` on `processInst` returned value), the interpreter considers the instruction executed and stops looking for a peripheral to execute it. If no peripheral executes the instruction, an error ("instruction unknown") is thrown.

The addition of new bytecode instructions remains outside of the peripheral modules. Although technically it is possible to extend the bytecode data type (`BC`) across separate modules, the amount of work necessary to do so outweighs the benefits it could bring. Currently, `BC`'s instance of the `iTask` type class is automatically derived. Clean can not automatically derive type classes of extended types. Therefore, if `BC` was extended, an instance of `iTasks` type class would have to be manually derived. Doing so would bring peripheral-specific code back to language core modules, going against the intend that drove the change to begin with.

The development that followed the changes described above proved that the separation of concerns regarding peripheral code improved mTask. Peripherals were added faster, with less code changes and less errors. Additionally, code maintainability increased substantially. Since peripheral code lays mostly in the same module, small changes can be performed faster and safer.

6.2.3 New Peripherals

Previously, the interpreted mTask supported the following peripherals: LEDs, LCD displays (for displaying numbers only), analog and digital pins. The standard Autohouse tasks required new peripheral support. The following peripherals were added to mTask: DHT22 temperature and humidity sensor, HCSR04 ultrasonic sensor, digital light sensor, Grove analog light sensor (P) V1.1, PIR and servo.

The digital light sensor, the Grove analog light sensor and the PIR did not require an external library to be used. Their data pin is connected to board pins and their values can be read using Arduino standard functions. An additional library was required to interface with the servo. The Arduino Servo library[6] was used. An additional library was also required to interface with the DHT22 sensor. Although there are libraries available out there, I decided to implement a small and simple one just for mTask: DHTino[7]. This choice was motivated by the limited amount of flash memory (32 KB) on the Arduino Uno. Existing libraries support many different sensors and have many features that would not be used by mTask. As a consequence, these libraries would take too much flash memory space. Guided by the same motivation, the Ultrino[8] library was created to interface with the HCSR04 ultrasonic sensor.

6.2.4 Device Requirements

Some tasks rely on certain peripherals to execute. For example, a task that regulates room temperature relies on a temperature sensor. Despite that, mTask did not provide a mechanism to determine whether a task is compatible with a device. The `Requirements` view was created to bring this feature to mTask. Its definition can be seen in Listing 6.5. `Requirement` is a type constructor with two phantom type variables: `a` and `b` [15]. These type variables are required by mTask type classes. `Requirement` is a wrapper around the device specification type `MTaskDeviceSpec`.

Given a mTask construct, this view will return the minimum device specification necessary to support that construct. This information can be used to determine whether a device matches the minimum specification for a task and therefore, if it is compatible with it. The `match` function (seen in Listing 6.5) does exactly that. Given an mTask program and a `Maybe MTaskDeviceSpec`, it yields whether the device and program are compatible.

[6] Arduino Servo Library. Available at `https://www.arduino.cc/en/reference/servo`. Accessed on September 10th 2018.

[7] DHTino on GitHub. Available at `https://github.com/matheusamazonas/DHTino`. Accessed on September 10th 2018.

[8] Ultrino on GitHub. Available at `https://github.com/matheusamazonas/Ultrino`. Accessed on September 10th 2018.

```
:: Requirements a b = Req MTaskDeviceSpec

match :: (Main (Requirements a b)) MTaskDeviceSpec → Bool

instance arith Requirements
instance UserLED Requirements
```

Listing 6.5: The `Requirements` view

Instances of mTask classes (including peripheral classes) are defined for `Requirement`. Therefore, given a mTask task, an application can filter the available devices based on whether they are compatible with it. The opposite is also possible: given a device, an application can filter tasks based on whether they are compatible with it.

6.2.5 Device Disconnection

By design, Autohouse should be robust regarding device disconnection (Section 5.4). Ideally, the system would detect a device disconnection and migrate the device's tasks to another suitable device. There were two challenges to tackle in order to implement this feature.

First, mTask does not recognize device disconnection for all of the device types it supports. Simulators never get disconnected. TCP devices throw an iTasks error when a disconnection is identified. This error is not caught by mTask and propagates upwards. Serial devices kill the application when disconnected. The library used by mTask to connect to Serial devices (CleanSerial[9]) halts execution when a device is disconnected.

In order to detect device disconnection, mTask had to be modified. If the device communication fails, the `channelSync` task (Section 4.2.5) should throw an exception[10]. TCP devices already throw an exception when communication fails and therefore require no change. Although simulators never disconnect from the system, simulating a disconnection would benefit testing. Hence, simulators were modified to support intentional disconnection. CleanSerial was modified to support device disconnection recognition.

Second, even if mTask recognizes device disconnection, it still can not communicate it to Autohouse. Ideally, mTask would communicate device disconnection through an error handler that would be provided by the application. Thus, the application would decide what task to perform in case of a disconnection. As seen in Section 4.2.5, the mTask library provides a single function to connect with a device: `withDevice`. This function is responsible (besides other tasks) to manage the connection to the device and therefore was the perfect place to insert an exception handler. An exception handler is a task that takes an error `String` as input. Listing 6.6 displays the type signature of the original `withDevice` along with its new version, named `withDevice'` here. If the application using mTask does not want to handle connection errors, the iTasks `throw` function can be used as the error handler. Doing so would propagate the exception upwards, emulating the behavior of `withDevice`.

```
withDevice  :: a (MTaskDevice → Task b)                    → Task b | channelSync a

withDevice' :: a (MTaskDevice → Task b) (String → Task ()) → Task b | channelSync a
```

[9] CleanSerial on GitLab. Available at `git@gitlab.science.ru.nl:mlubbers/CleanSerial.git`. Accessed on Semtember 8th 2018

[10] An iTasks task yields either a value or an exception. The iTasks standard library provides functions to create and handle exceptions.

```
throw :: e → Task a | iTask a & iTask e & toString e
```

Listing 6.6: Change in mTask to support a device disconnection handler

Consequently, mTask recognizes and provides an exception handler for device disconnection. Autohouse uses this feature to detect unit disconnection and thus automatically migrate tasks from the disconnected device to a suitable one.

6.2.6 Simulator Improvements

The simulator (Section 4.2.4) proved to be an essential tool during the development of Autohouse. Although, it was clear that it could be improved to ease debugging and testing of the application.

Sometimes, the developer might want to debug a task and inspect it closely. The simulator's manual mode is adequate for such usage, but it might be a bit cumbersome to use. Specially with large tasks, stepping over each program instruction becomes a rather tedious and inefficient process. With that in mind, the simulator was extended to support breakpoints on bytecode instructions. Tools to add and to step over breakpoints were added to the simulator UI. When executing a task, the simulator goes through its bytecode instructions, checking if there are breakpoints on each instruction before executing it. If an instruction has a breakpoint, execution waits for user input (by clicking on "step over") to continue. At any point, the user is able to edit breakpoints.

The ability to simulate peripheral values is crucial for program testing in mTask. Tasks often rely on peripheral values and therefore can only be thoroughly tested if peripheral values can be simulated. Although, the simulator did not have such feature. The development of Autohouse showed how necessary this feature is for mTask development. Hence, simulation of peripheral values was incorporated into the simulator. Values can be manually set via the simulator UI, similarly to breakpoints.

6.3 Limitations of mTask

Some of the limitations of mTask that surfaced during the development of Autohouse were not overcome. First, SDSs can not be removed from a device. There is no message in mTask's communication protocol (Section 4.2.2) to request SDS deletion. Since an SDS is not bound to a task, once it is sent to a device, it lives there indefinitely. As a consequence, a device can accumulate unused SDSs over time, possibly filling the device's memory with dangling SDSs. Ideally, SDSs would always be bound to a task. Thus, they would be removed along with its task, eliminating dangling SDSs.

Second, there is a communication loop on SDS updates. The mTask library automatically synchronizes SDSs between server and devices. Whenever a device publishes an SDS value, a message is sent to the server, which updates the actual SDS. Also, the server observes an SDS and whenever it is modified, it sends an update message to every device that uses that SDS. Hence, the server ensures SDS synchronization. A problem arises because the server is not aware of who is updating an SDS. Therefore, it might send an update message to the same device that published the SDS value, creating a communication loop. This behavior is not desirable because it generates unnecessary communication between the server and the devices. Additionally, it might create unexpected behavior. Ideally, the server would identify who is updating the SDS and avoid sending an update message to the device that trigerred it.

Additionally, mTask does not communicate whether a task sent to a device was acknowledged. Ideally, a call to `liftmTask` would communicate task acknowledgment using a callback handler (similarly to the connection error handler in Section 6.2.5). The application might want to wait

for the task acknowledgment to continue. For example, Autohouse could wait for a task acknowledgment to store the task data in the unit SDS. Since it can not detect task acknowledgment, it stores the task data before actually sending the task. As a consequence, if a task fails to be acknowledged, an invalid task will live in the device's task list.

Finally, mTask does not support floating-point arithmetic. Some sensors data (e.g. DHT22 temperature and humidity sensor) is represented using floating-points and could not be directly translated into mTask bytecode values. As a workaround, the Arduino client uses integers to represent floating-points, with a precision of two decimals.

6.4 Task Migration

In case of device disconnection, the application should migrate the unit's tasks to another suitable one. The mTask library offers means to connect and send task to devices, but not to manage them. The application using mTask is responsible for device management. Therefore, automatic task migration was implemented entirely in Autohouse and required no changes to mTask whatsoever.

First, the feature behavior was defined. Although automatic task migration might be helpful, not all tasks should be automatically migrated. For example, some tasks that are suitable for a hallway unit (e.g. motion-activated light switch) might not be suitable for a bedroom unit. Therefore, different migration strategies were created. These strategies are represented in the `Migration` ADT, seen in Listing 6.7.

```
:: Migration = DoNotMigrate | SameRoom | AnyRoom
```

Listing 6.7: Task migration strategies of Autohouse

Tasks with `DoNotMigrate` migration strategy never migrate to another unit. Tasks with the `SameRoom` strategy migrate only to units within the same room of its original unit. Tasks with `AnyRoom` strategy migrate to any other unit in the smarthome. When a task is being migrated, other units are checked for compatibility (based on the task's strategy, following no particular order) and once a compatible unit is found, the task is sent to it. If no compatible unit is found, the task is not migrated. The Autohouse user is responsible for determining a task's migration strategy when sending it to a unit.

The application has to save enough task data to migrate a task in case of a device disconnection. The mTask's `liftmTask` function (Section 4.2.5) is used to send tasks to devices. Besides the device itself, this function takes a task interval and an mTask task as input. Therefore, this is all the information the application needs to send a task to a device. Autohouse's default tasks have a unique id that can be used to retrieve the task from the default task list. Thus, the application should only need to store the task index and its interval (`MTaskInterval`) in order to migrate it.

But storing the task id and interval is not enough. Some Autohouse tasks require user-provided arguments to work. For example, a user must provide an initial target temperature to a thermostat task before sending it to a device. If such a task is being migrated, the application should be able to restore the task arguments as well. Autohouse stores task arguments along with the task index and interval. All the information necessary to migrate an Autohouse program lays in the `ProgramInstance` data type, seen on Listing 6.8.

```
:: ProgramInstance = { pIx   :: Int,
                     , pArgs :: [Dynamic],
                     , pInt  :: MTaskInterval,
```

```
                pMig  :: Migration }
```

Listing 6.8: Task migration data

A unit contains a list of `ProgramInstance`s. Each list element represents one task running on the unit and can be used to migrate its respective task to a new device. If a unit disconnects, the application goes through the unit's `ProgramInstance` list and migrates the tasks accordingly. Automatic task migration was tested using simulators and physical devices and it worked as expected.

Chapter 7

Related Work

7.1 mTask

Recent research has been conducted on mTask. A new, task-based version of it has been proposed [13]. On this version, the imperative language has been replaced by a functional one. This new version was not used during the research reported in this document because its implementation was not available on time.

The usage of programming languages to interface with microcontrollers has been a subject of research. Firmata[1] is a protocol to control microcontrollers. Its messages follow the MIDI message format and model mostly commands on analog and digital input and output pins. There is a client-side implementation for the Arduino[2] and host-side implementations for many programming languages, including a Haskell implementation to communicate with Arduinos called hArduino[3]. Since Firmata is a protocol and not a programming language, full applications can not be built using Firmata solely. Other tools are built on top of it.

The Haskino library enables Arduino programming using Haskell [11]. The library is available in two different flavors. The first one is based on hArduino (and consequently, on Firmata) and requires the Arduino to maintain a serial connection with the host. On this approach, most of the program evaluation is executed on the host and only I/O commands run on the client. The second approach drops Firmata and uses its own communication protocol. The client is more independent and can execute more elaborate commands, including control flow constructs. In contrast with the first approach, it presents a lower communication overhead. In addition, programs can be written to the Arduino's EEPROM, allowing standalone execution. When compared to mTask, both flavors of Haskino depend heavily on the server.

Some research has been made on generating C/C++ code for microcontrollers from high level languages. Ivory is an EDSL embedded into Haskell that generates safe embedded C code [12, 8]. By design, the generated code is memory safe and free from common errors and undefined behaviors. It uses Haskell's type system (with some GHC extensions) to avoid errors like array indexing out of bounds, main loop function with return statements and dangling pointers. Additionally, it prohibits (by design) some standard C features that might generate unsafe code. Ivory was used on the development of the SMACCPilot, a high-assurance autopilot system for quadcopter Unmanned Air Vehicles (UAV). Unlike mTask, Ivory does not support

[1] Firmata Protocol Documentation. Available at `https://github.com/firmata/protocol`. Accessed on August 10th 2018.

[2] Firmara Arduino. Available at `https://github.com/firmata/arduino`. Accessed on August 10th 2018.

[3] hArduino. Available at `http://leventerkok.github.io/hArduino`. Accessed on August 10th 2018.

dynamic uploading of new tasks.

The `frp-arduino` library[4] implements the Functional Reactive Programming (FRP) paradigm as an EDSL embedded in Haskell. Programs in the EDSL can be compiled to Arduino C code which can be uploaded to Arduino boards. Juniper[5] is another FRP language for the Arduino. It is a standalone programming language that transpiles to Arduino C++.

Additionally, some programming language interpreters were ported to microcontrollers. Espruino[6] is a JavaScript interpreter for microcontrollers. It officially supports only proprietary boards but other microcontrollers such as the ESP8266 and the members of the STM32 family are supported by the community. Due to hardware limitations, none of the Arduino boards are supported. Espruino's official website lists many projects that were built using it, including home automation applications.

Micropython[7] is a lean implementation of the Python interpreter and parts of its standard library for microcontrollers. Its main target device is the proprietary *pyboard*. Given that it requires at least 16KB of RAM, it is not compatible with most Arduino boards. It is compatible with microcontrollers of the STM32 family. Many projects (including home automation) were developed using Micropython and *pyboards*.

Finally, the programming of microcontrollers dynamically (without the need to plug it to a computer) is a well known practice. For example, the ESP8266[8] Wi-Fi module supports Over-the-Air (OTA) programming. The Arduino Uno Wi-Fi[9] is a version of the Arduino Uno board that contains an ESP8266 module and supports OTA programming natively via the Arduino IDE. It is important to note that although OTA enables dynamic programming of microcontrollers, it differs from mTask's dynamicity. On OTA programming, the device memory is reset when a new program is loaded. On the dynamic version of mTask, the device's memory and the tasks running on it are unaffected.

7.2 Autohouse

Autohouse is a home automation application built with the intent to assess mTask's capabilities and was not meant to be commercialized. Therefore, when compared to existing commercial home automation systems, mTask supports less platforms, offers less features and does not focus on some aspects (e.g. security, performance, user experience). Although, there are some fundamental differences between these applications and Autohouse. A brief comparison follows.

The openHAB[10] is an open-source home automation integration platform. Instead of controlling devices running a custom firmware, openHAB integrates existing automation system from different manufacturers. Therefore, Autohouse and openHAB operate in different abstraction levels.

Home Assistant[11] is an open-source automation system that supports many automation platforms including Arduino. Users can configure automation tasks using YAML files. Although it

[4]FRP on Arduino. Available at `https://github.com/frp-arduino/frp-arduino`. Accessed on August 10th 2018.

[5]Juniper Programming Language. Available at `http://www.juniper-lang.org/index.html`. Accessed on August 11th 2018.

[6]Espruino. Available at `https://www.espruino.com`. Accessed on August 10th 2018.

[7]Micropytohn. Available at `https://micropython.org`. Accessed on August 10th 2018.

[8]ESP8266 Overview. Available at `https://www.espressif.com/en/products/hardware/esp8266ex/overview`. Accessed at August 11th 2018.

[9]Arduino Store - Arduino Uno Wi-Fi. Available at `https://store.arduino.cc/arduino-uno-wifi`. Accessed on August 11th 2018.

[10]openHAB. Available at `https://www.openhab.org`. Accessed on September 19th 2018.

[11]Home Assistant. Available at `https://www.home-assistant.io`. Accessed on September 19th 2018.

supports Arduino, the server uses Firmata protocol (Chapter 7) to control the devices. Therefore, evaluation of automation tasks is performed on the server, not on the clients. As a consequence, if the connection between server and clients fails, the devices stop performing tasks. In Autohouse, devices keep executing tasks if communication is interrupted.

Blynk[12] and Thinger.io [13] are IoT platforms in which devices (e.g. Arduino, Mbed, Raspberry Pi[14]) can be controlled via iOS and Android apps. Unlike in Autohouse, new tasks can not be sent to the devices dynamically.

As seen above, mTask's microcontroller support and dynamic nature brings a set of features to Autohouse that none of the analyzed home automation systems possesses.

[12]Blynk. Available at `https://www.blynk.cc`. Accessed on September 19th 2018.
[13]Thinger.io. Available at `https://thinger.io`. Accessed on September 19th 2018.
[14]Raspberry Pi. Available at `https://www.raspberrypi.org`. Accessed on September 19th 2018.

Chapter 8

Conclusion

8.1 Discussion and Future Work

8.1.1 mTask

The Autohouse application was successfully developed using mTask, but some aspects still have to be analyzed. The tests performed during development were limited to five Arduino boards running a maximum of three simultaneous tasks for a maximum duration of one hour. It is unknown how mTask performs running more than five simultaneous devices. Given that the number of devices in IoT applications can escalate quickly, it is important to assess how mTask escalates in regard to the number of connected devices. Similarly, given that IoT applications often run continuously, it would be important to perform tests with tasks running for longer periods of time. Another characteristic of IoT solutions is lower power consumption [21, 18, 22]. Although, no power consumption analysis of mTask applications was performed. Assessing mTask's power consumption, scalability and behavior over long periods is suggested as future research.

Ideally, the limitations of mTask described in Section 6.3 should be eliminated. First, SDSs and tasks should be sent in a bundle. Thus, they would be removed altogether, eliminating dangling SDSs. Second, mTask would avoid sending update messages to the exact same device which triggered the SDS update. Lastly, the mTask library would provide callbacks to signal task acknowledgment. Solving these problems is also proposed as future research.

The Raspberry Pi[1] is a compact ARM computer often used in IoT projects both as a device and as a server. Since it is capable of running an iTasks core, it should be able to run an mTask server [19]. Also, since the Pi hardware is Linux compatible, it might host an mTask POSIX client. Testing with the Raspberry Pi is suggested as future research.

The mTask EDSL was created to bring IoT devices to the iTasks environment. Its language is imperative and the system is not task-centred. Also, tasks cannot be combined as in iTasks, limiting mTask's expressiveness. Naturally iTasks and mTask differ on some aspects, but ideally, the gap between them would be minimal, both in syntax and semantics. Current research has been performed to bridge this gap. A functional, task-centred version of mTask has been proposed [13]. Parallel and sequential task combinators were introduced, improving mTask's expressiveness. Due to time constraints, this version of mTask could not be used in this research. Assessing the abilities of this new version of mTask is proposed as future research.

[1] Raspberry Pi. Available at `https://www.raspberrypi.org`. Accessed on September 19th 2018.

8.1.2 Autohouse

Although Autohouse is not the research focus, it would be interesting to extend the application to push the boundaries of mTask.

Currently, devices have no information about what its peripherals represent. For example, a servo can be used to control curtains or to lock a door. The user might know what which peripheral represents when sending a new task to a device, but the migration algorithm does not. As a consequence, a task might migrate to a compatible device with a peripheral that controls a different object than the intended one. For example, a task that uses a servo to automatically close curtains might migrate to a device that uses its servo to lock a door. Ideally, the application user would attribute tags to its peripherals. The migration algorithm would take that information into consideration when migrating tasks.

8.2 Conclusion

The research reported in this documented tested mTask's ability to develop real-life IoT applications. The research question was tackled by example: the Autohouse application intended to assess mTask's capabilities. The application is a home automation system that allows users to dynamically manage automation tasks running on devices spread across different rooms.

Limitations of mTask surfaced during the development of Autohouse. Some limitations were overcome by changing the mTask and CleanSerial libraries. Task variables were added to the language. Device disconnection recognition was implemented, allowing the application to automatically migrate tasks when a device is lost. A new view was added to the EDSL which generates minimum device requirements for a mTask task. This view can be used to filter available devices based on whether they support a given task. Six new peripherals were added to the mTask language and to the Arduino client. Peripheral code was restructured, easing the addition of new peripherals, increasing code maintainability and bringing a better separation of concerns between the language core constructs and peripheral constructs. Finally, the simulator for the interpreted mTask was modified to support the setting of peripheral values and breakpoints, which improved testing and debugging considerably.

Other limitations could not be overcome during this research. SDSs are never removed from devices and live there indefinitely. There is an unwanted communication loop between devices and server whenever a device publishes an SDS. The mTask library does not communicate task acknowledgment. Although these limitations were not overcome, they did not stop the development of Autohouse.

The mTask EDSL and library were successfully used to develop a real-life IoT application: the home automation system Autohouse. Some of the limitations unearthed during the development process were overcome and some remain. Finally, it is clear what the next steps to improve mTask are.

Bibliography

[1] P. Achten, P. Koopman and R. Plasmeijer. "An Introduction to Task Oriented Programming". In: *Central European Functional Programming School: 5th Summer School, CEFP 2013, Cluj-Napoca, Romania, July 8-20, 2013, Revised Selected Papers*. Ed. by V. Zsók, Z. Horváth and L. Csató. Cham, Switzerland: Springer International Publishing, 2015, pp. 187–245.

[2] *Arduino - Home*. URL: `https://www.arduino.cc` (visited on 06/08/2018).

[3] T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer and M. J. Plasmeijer. "Clean — A language for functional graph rewriting". In: *Functional Programming Languages and Computer Architecture*. Ed. by G. Kahn. Berlin, Heidelberg: Springer, 1987, pp. 364–384.

[4] J. Carette, O. Kiselyov and C.-c. Shan. "Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages". In: *Journal of Functional Programming* 19.5 (Sept. 2009), pp. 509–543.

[5] J. Cheney and R. Hinze. *First-class phantom types*. Tech. rep. Cornell University, 2003.

[6] J. van Diggelen, W. Post, M. Rakhorst, R. Plasmeijer and W. van Staal. "Using Process-Oriented Interfaces for Solving the Automation Paradox in Highly Automated Navy Vessels". In: *Active Media Technology*. Ed. by D. Ślęzak, G. Schaefer, S. T. Vuong and Y.-S. Kim. Cham, Switzerland: Springer International Publishing, 2014, pp. 442–452.

[7] L. Domoszlai, B. Lijnse and R. Plasmeijer. "Parametric Lenses: Change Notification for Bidirectional Lenses". In: *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages*. IFL '14. ACM, 2014, 9:1–9:11.

[8] T. Elliott et al. "Guilt Free Ivory". In: *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*. Haskell '15. Vancouver, BC, Canada: ACM, 2015, pp. 189–200. ISBN: 978-1-4503-3808-0.

[9] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce and A. Schmitt. "Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem". In: *ACM Trans. Program. Lang. Syst.* 29.3 (May 2007).

[10] *Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016*. 2017. URL: `https://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-31-percent-from-2016` (visited on 04/08/2018).

[11] M. Grebe and A. Gill. "Haskino: A Remote Monad for Programming the Arduino". In: *Practical Aspects of Declarative Languages*. Ed. by M. Gavanelli and J. Reppy. Cham, Switzerland: Springer International Publishing, 2016, pp. 153–168.

[12] P. C. Hickey, L. Pike, T. Elliott, J. Bielman and J. Launchbury. "Building Embedded Systems with Embedded DSLs". In: *SIGPLAN Not.* 49.9 (Aug. 2014), pp. 3–9.

[13] P. Koopman, M. Lubbers and R. Plasmeijer. “A Task-Based DSL for Microcomputers”. In: *Proceedings of the Real World Domain Specific Languages Workshop 2018*. Vienna, Austria: ACM, 2018, 4:1–4:11.

[14] P. Koopman and R. Plasmeijer. “A Shallow Embedded Type Safe Extendable DSL for the Arduino”. In: *Trends in Functional Programming*. Ed. by M. Serrano and J. Hage. TFP 2015. Cham, Switzerland: Springer International Publishing, 2016, pp. 104–123.

[15] D. Leijen and E. Meijer. “Domain Specific Embedded Compilers”. In: *Proceedings of the 2Nd Conference on Domain-specific Languages*. DSL ’99. Austin, Texas, USA: ACM, 1999, pp. 109–122.

[16] B. Lijnse, J. M. Jansen and R. Plasmeijer. “Incidone: A Task-Oriented Incident Coordination Tool”. In: *Proceedings of the 9th International Conference on Information Systems for Crisis Response and Management*. ISCRAM’12. Vancouver, BC, Canada: Simon Fraser University, 2012.

[17] M. Lubbers. “Task Oriented Programming and the Internet of Things”. Master’s thesis. Nijmegen: Radboud University, 2017.

[18] D. Miorandi, S. Sicari, F. D. Pellegrini and I. Chlamtac. “Internet of things: Vision, applications and research challenges”. In: *Ad Hoc Networks* 10.7 (2012), pp. 1497–1516.

[19] A. Oortgiese, J. van Groningen, P. Achten and R. Plasmeijer. “A Distributed Dynamic Architecture for Task Oriented Programming”. In: *Proceedings of the 29th Symposium on the Implementation and Application of Functional Programming Languages*. IFL 2017. Bristol, United Kingdom: ACM, 2017, 7:1–7:12.

[20] R. Plasmeijer, B. Lijnse, S. Michels, P. Achten and P. Koopman. “Task-oriented Programming in a Pure Functional Language”. In: *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming*. Leuven, Belgium: ACM, 2012, pp. 195–206.

[21] P. Ray. “A survey on Internet of Things architectures”. In: *Journal of King Saud University - Computer and Information Sciences* (2016).

[22] L. Sanchez et al. “SmartSantander: IoT experimentation over a smart city testbed”. In: *Computer Networks* 61 (2014), pp. 217–238.

Glossary

Android Linux-based mobile operating system developed by Google.

Arduino An open-source electronics prototyping platform.

Autohouse Home automation application developed using mTask and iTasks.

C A general purpose, structured imperative language.

C++ A general purpose, object-oriented imperative language.

Clean A general purpose, pure and lazy functional programming language.

iOS Operating system created by Apple to its proprietary hardware (smartphones, tables) .

iTasks A TOP implementation hosted in Clean.

Mbed Platform and operating system for IoT devices based on ARM Cortex-M microcontrollers.

Microcontroller A compact, integrated circuit containing a small computer, plural=microcontrollers.

mTask An EDSL embedded in Clean to control IoT devices in iTasks.

servo Servomotor. A rotary motor with precise control of angular position.

YAML Human friendly data serialization language.

Acronyms

ADT Algebraic Data Type.

ARM Advanced RISC Machine.

DSL Domain Specific Language.

EDSL Embedded Domain Specific Language.

EEPROM Electrically Erasable Programmable Read-Only Memory.

FRP Functional Reactive Programming.

GADT Generalized Algebraic Data Type.

GHC Glasgow Haskell Compiler.

GPL General Purpose Language.

HTML Hypertext Markup Language.

IDE Integrated Development Environment.

IoT Internet of Things.

IP Internet Protocol.

LAN Local Area Network.

LCD Liquid Crystal Display.

LED Light-Emitting Diode.

MIDI Musical Instrument Digital Interface.

OTA Over-the-Air.

PIR Passive Infrared Sensor.

POSIX Portable Operating System Interface.

RISC Reduced Instruction Set Computer.

SDS Shared Data Source.

SPP Serial Port Protocol.

TCP Transmission Control Protocol.

TOP Task Oriented Programming.

UAV Unmanned Air Vehicles.

UI User Interface.

USB Universal Serial Bus.

VHDL VHSIC Hardware Description Language.

VHSIC Very High Speed Integrated Circuit.

List of Listings

List of Figures