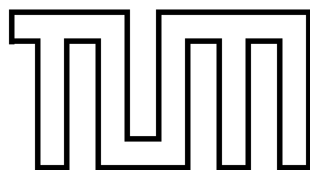


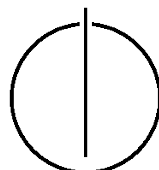
FAKULTÄT FÜR INFORMATIK

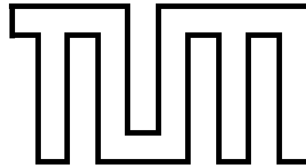
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis
in Informatik

Development of a concurrent language featuring a type system based on Boolean implications

René Neumann





FAKULTÄT FÜR INFORMATIK

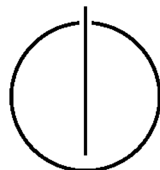
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis
in Informatik

Development of a concurrent language featuring a type system based on Boolean implications

Entwicklung einer nebenläufigen Sprache mit einem auf boolschen Implikationen basierten Typsystem

Author:	René Neumann
Supervisor:	Prof. Dr. Helmut Seidl
Advisor:	Dr. Axel Simon
Date:	April 15, 2011



Ich versichere, dass ich diese Masterarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

I assure the single handed composition of this master's thesis, only supported by declared resources.

München, den 3. Juli 2012 René Neumann

Acknowledgments

This thesis would not be possible without the discussions with Axel Simon. He provided helpful insights and the proper hints that often enough prevented this system from falling apart.

Abstract

The well-known Hindley-Milner type system has several shortcomings. By presenting a new type system together with a type inference algorithm we try to address some of these shortcomings without losing the advantages of Hindley-Milner.

The type system uses constructor sets as types and therefore allows an easy approach to subtyping. Furthermore, by extending the basic function constructor, it allows functions to have multiple resulting types depending on the values of the arguments.

The type inference algorithm is split up in three phases: First automata construction to model the correct branching behavior. Then collecting the types of each expression and finally creating a set of implications between the different types. These implications allow to follow the flow of types through the program.

Additional to the type system, a process-based concurrent language to work with the type system is designed and presented in syntax and semantics.

Contents

Acknowledgements **vii**

Abstract **ix**

1. Introduction **1**

2. Background **3**
2.1. Programming languages 3
2.2. Operational Semantics 3
2.3. Type Systems 4
2.4. Related Work 5

3. Grips – the language **7**
3.1. Language design and syntax 7
3.1.1. Lambda calculus 7
3.1.2. Constructors and patterns 8
3.1.3. Processes 10
3.1.4. Futures 12
3.2. Operational Semantics 13

4. Development of the type system **21**
4.1. Description of the types 22
4.2. Automata construction 24
4.2.1. Notation 25
4.2.2. Edge refinement 26
4.2.3. The algorithm $\mathcal{A}$ 29
4.2.4. Mapping virtual nodes 34
4.2.5. Correctness check 36
4.3. Constructor collection 39
4.3.1. Typing Rules 40
4.3.2. Combining the Types / Modularity 43
4.4. Implication analysis 44
4.4.1. Notation 45
4.4.2. Implication Rules 47

5. Conclusion **51**

A. Grips language **53**
A.1. Syntax 53
A.2. Syntactic sugar 54

Bibliography **55**

List of Figures

3.1. Example, showing multiple future passing . . . 14
3.2. Examples of `createprocs` uses . . . 18
3.3. Transformation rules . . . 19

4.1. Pattern definition . . . 22
4.2. Example program . . . 25
4.3. Example automaton . . . 25
4.4. Code example of nested refinement . . . 27
4.5. Nested automaton . . . 29
4.6. Creating the automaton of an assignment . . . 32
4.7. Creating the automaton of a match . . . 34
4.8. Divergence in virtual nodes . . . 36
4.9. Example program . . . 45
4.10. Example program after substituting types . . . 45

List of Algorithms

4.1. Creating a new label 27
4.2. Label printing 27
4.3. Label refinement 28
4.4. Explicit behavior of the preserving call 28
4.5. Match Rule 35
4.6. Extended transition relation $\hat{\hat{\delta}}$ 37
4.7. Mapping virtual nodes 37
4.8. Resolving one virtual node 38

1. Introduction

The work of Hindley [Hin69], Milner [Mil78], Damas [DM82] and Mycroft [Myc84] laid the foundations for the modern type systems used in almost all current programming languages.

Unfortunately, the type system these authors describe has several short-comings: Some constructs of a language might be semantically legal but cannot be typed without the annotation of explicit type information. It indeed does not allow the static inference of the most general type of a function in isolation from other functions in the current program [Wel02]. This, amongst other consequences, leads to problems in regard to useful error messages [Chi01].

Additionally, the Hindley-Milner system requires the resulting type of a function to be independent from the value of an argument. Hence it is impossible to construct and type a function, which evaluates to a type α in case it receives some value A and to type β in all other cases. This restriction might be seen as an obstacle when modeling programmatic functions as processes rather than mathematical functions. Such a process receives messages and behaves according to those messages, like for example a process `sum`, that sums up all integers sent to it until it receives a special token (here: A) that makes it return the calculated sum:

```
sum’ acc x = x | A -> acc -- type: Int
               | _ -> sum’ (acc + x) -- type: (Int ->)* A -> Int
sum = sum’ 0
```

Here it is desired to allow type safety on the one hand, but leaving the corset that the Hindley-Milner system imposes[1] on the other hand.

In this thesis we will develop a type system that tries to tackle these problems: It defines types as a set of constructors, where the function type does not stand special and is handled as a constructor, too. Therefore it is possible for a single function to have a type that is a set of function constructors. Additionally, the function constructors were extended to also hold the pattern the argument has to match. Hence a function constructor $x \xrightarrow{A} y$ expects x to match the pattern A and thus is different from the function $x \xrightarrow{B} y$. Now,

[1] Wrapping the resulting types in explicit choice datatypes like `Either` in Haskell can only be seen as some sort of workaround and not as a general solution.

it is possible to have the behavior of result types depending on the argument's value as described beforehand.

The other important point, namely modularity, is resolved by using an approach to type inference, where the control flow and the flow of types is modeled. This is done as a step for each function in isolation and only at the time all modules are connected (*linking*) the inter-function flows are evaluated and inferred.

There exists a further weakness: The Hindley-Milner system does not allow polymorphic functions as first-class objects and thus disallows passing polymorphic functions as arguments. This, for instance, leads to type errors for the following Haskell [PJ03] snippet:

```
let f x = (x 1, x 'a')
in f id
```

This weakness may or may not yet be removed by the type system presented in this work – it depends on the definition of polymorphicy (cf. Sec. 4.1). Small modifications of the type inference algorithms should allow the use with a type system containing type variables. Such a type system is without oppositon polymorphic.

This thesis is organized as follows: In Chapter 2, we give a short introduction into type systems and semantics. We then present a language this type system has been designed for in Chapter 3, and finally, in Chapter 4, we describe its type inference with its different phases of automata construction, constructor collection and implication generation.

This thesis will not present any proofs of soundness or completeness of the provided algorithms, but instead will concentrate on the design of the language, its semantics, and the definition of the type inference. As the influence between these three parts is mutual, the language itself is regarded as part of the core of this work.

2. Background

In this chapter we will give a short overview about the backgrounds of programming languages, their semantics and type systems.

2.1. Programming languages

Programming languages can be classified by different paradigms. One is whether they work in an imperative, functional or logical fashion. Imperative languages like *C* [KR88] or *Java* [GJSB05] are characerized by algorithms that "describe computation in terms of state-transformation operations" (cf. [Rey98]). These languages make excessive use of memory cells, which are updated destructively during the algorithms. Functional languages like *SML* [MTHM97] or *Haskell* [PJ03] on the other hand express the program as a mathematical function, which is evaluated. Variables are immutable here. The third species, logical languages like *Prolog* [SS86], see their programs as a series of logical formulas. Results for these programs are created by finding solutions to the set of formulas.

From these three, the imperative and the functional approach are the most common ones, but functional languages are preferred in academia due to their tight coupling to mathematics and the avoidance of side-effects. This allows for an easier theoretical reasoning about the properties of a language.

Another property to differentiate programming languages is their ability for concurrency: Non-concurrent languages only allow a single thread of execution (*serial computation*), though they might have techniques to have multiple *programs* running in parallel and communicating with each other. Concurrent languages on the other hand allow to run multiple threads either explicitly or implicitly inside one program.

2.2. Operational Semantics

The syntax of a language does not describe the correct behavior of a program. So there is the need to formalize the way a program expression behaves. This is done by strictly laying down the semantics of that language.

These semantics can be represented in different terms. One of them is the style of *operational semantics*, which specifies the behavior of the language in terms of states. The current configuration of the states is determined by the syntax of the language: For each possible syntactic term in the language, a rule is created that shows under which premisses a configuration is transformed into another one.

There exist two different kinds of operational semantics: *Small-step operational semantics* or *structural operational semantics* were first introduced by Plotkin [Plo81] and model the behavior of a term step-by-step. That is, given a current command c in state s, they describe under which circumstances this can be evaluated into the command c' with state s'.

On the other hand, there is also *big-step operation semantics* or *natural semantics* [Kah87], which only describe the final state that is reached by a given command.

Besides the operational semantics, the semantics might also be specified using *denotional semantics* [SS71], where a more abstract view is taken by stating the meaning of a term as some mathematical object. These allow a complete abstraction from the program itself into some mathematical domain and therefore to reason in this domain. Unfortunately, this entails some prior work in finding the mathematical domain to be used. Quoting Tofte [Tof88],

> It seems a bit unfortunate that we should have to understand domain theory to be able to investigate whether a type inference system admits faulty programs.

we prefer the aforementioned small-step operational semantics approach.

2.3. Type Systems

> A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.

This definition is given by Pierce in [Pie02] and essentially condenses the reason for the existance of type systems: To provide means to catch errors and generally assert that the program will not fail.

In typed languages, each expression in the program obtains a *type*, which describes the set of values this expression may result in. The type system then consists of a set of rules which describe how types have to match.

A main difference between type systems is whether they are *statically* or *dynamically* typed. The first one assigns types at compile time, while the latter does so during runtime. Another difference lies between *monomorphic* and *polymorphic* type systems. In monomorphic

systems, expressions have exactly one type, while they have multiple types in polymorphicly typed languages (see the end of Section 4.1 for a discussion on different definitions of polymorphicy).

Further, type systems can be grouped by whether they use *explicit* or *implicit* typing (or a mixture of both): Explicit typing requests, that the programmer annotates the type information to each variable and function inside the program. This information is then used to check whether the program is type safe. In implicitly typed programs, the programmer omits all type information and the type system has to infer it.

Type systems are a broad research area and therefore we do not intend to give an overview about the different kinds and abilities of those systems. Instead we refer to Cardelli [Car97], who gives an extensive overview and to Pierce's book [Pie02], which provides a comprehensive coverage of this topic.

2.4. Related Work

The approach taken in this thesis seems to be a novel one. We were not able to find publications which go a similar way. Nevertheless, the work by Marlow and Wadler [MW97] was used as an inspiration during the development, but at the current shape, the two systems differ largely. So does their system omit first class functions and interprocess communications – both two essential requirements for our work.

The idea of adding constraints to types is not a new one, and has already been described for example by [XP99] and [SP07]. But both approaches differentiate between data-types and types, i.e. a certain type is described by different data constructors (e.g. the type `list` by the two constructors `Nil` and `Cons`) and the constraints are added to these data constructors. This is not applicable for our approach where types are defined as sets of constructors and the constraints are part of the whole type.

3. Grips – the language

The type system, which provides the largest part of this thesis, utilizies new concepts. To use these concepts in a practical environment and not only theoretically on paper, we created a language to go with this system. This language, called “Grips”, will be used throughout the coming chapters to describe the features of the type system. In this chapter, we are going to present the language itself: we describe the design, syntax and semantics of Grips.

3.1. Language design and syntax

As described in section 2.1, there are different classes of paradigms for programming languages. It was outlined there, that the class of functional languages has the property of being simple to reason about. Therefore Grips is designed to be of functional nature. Moreover, with the rise of current multiprocessor systems and the existance and usage of large networks, concurrency has become an important property and hence Grips will also be concurrent. These properties make it similar to *Erlang* [Arm03], but not the same: For instance Erlang has explicit spawning of processes, while this is an implicit action in Grips. Also Erlang has advanced features like replacing program modules at runtime, which are not part of our language.

The raison d’être of this language is, as was mentioned, to have an application for the type inference algorithm of this thesis. Therefore this language is implicitly typed and has no syntax for specifying type information.

The functionality and syntax of Grips will be presented in four steps: Basic lambda calculus, constructors and patterns, processes, and finally futures. The complete syntax can be found in appendix A. This syntax does implement only the core functionality, and more complex constructs, which are built using these core elements, are added as *syntactic sugar*. This syntactic sugar will be used throughout the code examples where appropriate and its rules are also outlined in the aforementioned appendix.

3.1.1. Lambda calculus

We will describe the functionality of this language first without the aspect of concurrency, as this is not needed for the general understanding: From the basic building blocks it is

a simple lambda-calculus with let-bindings (see [Bar90] for detailed information), as can be found in most functional languages like Haskell or SML. This is enriched with another common feature – pattern matching – to allow branching depending on the content of a variable. Together, these elements give rise to the following part of the core syntax, which will be completed during the coming sections:

$$
\begin{array}{rll}
expr ::= & \texttt{skip} & \text{(Empty expression)} \\
\mid & \texttt{end} & \text{(Empty expression \#2)} \\
\mid & var & \text{(Variable call)} \\
\mid & \lambda\, var.\, expr & \text{(Abstraction)} \\
\mid & expr_1\ expr_2 & \text{(Application)} \\
\mid & \texttt{let}\, var = expr_1\, \texttt{in}\, expr_2 & \text{(Let-Binding)} \\
\mid & \texttt{letrec}\, var = expr_1\, \texttt{in}\, expr_2 & \text{(Recursive Let-Binding)} \\
\mid & ?var\, |_{m_1}\, var_1@varpat_1 \to expr_1 & \\
 & \quad \vdots \qquad ;\ expr & \text{(Pattern Matching)} \\
 & \quad |_{m_n}\, var_n@varpat_n \to expr_n &
\end{array}
$$

Here, *var* is any variable name and *varpat* is a pattern with variables and will be explained during the next section.

The two different empty expressions – `skip` and `end` – behave the same, except when used inside pattern matching. This difference and the detailed semantics of the pattern matching will be also explained in coming sections.

We further define the sets $\mathbb{E}$ and $\mathbb{V} \subset \mathbb{E}$ to be the sets of expressions and variables, respectively. Here, $\mathbb{E}$ are all expressions that adhere to the grammar given with *expr*, while $\mathbb{V}$ is defined via *var*.

Also all expressions directly at top-level inside a source file are seen as being in global scope, i.e. they are visible everywhere. Different source files are at the moment seen as one global scope with a flat namespace. Note though, that the Grips core language does not define syntax for including or importing other modules.

3.1.2. Constructors and patterns

With the constructs above, there is no way of defining data. To achieve this task, Grips uses constructors of the form $C\ expr_1 \ldots expr_n$. These constructors are *the only way* to work on data. Therefore also basic datatypes like integers need to be wrapped inside a constructor (e.g. $Int\ 2$), where the argument needs to be handled by a given special set of functions.

Otherwise it would not be possible to describe a function `plus` $:: Int \to Int \to Int$, as the plain integer wrapped in the *Int* constructor cannot be retrieved. This is a known approach and is for example also used in *Python* [VR03, pyt11] and furthermore includes the definition of certain functions outside the language's own semantics.

There is currently also no syntax to describe constructors, i.e. specify their arity and types of arguments. Instead they are simply used throughout the program and the compiler asserts that all uses of a certain constructor have the same number and types of arguments. Hence it is not possible for constructors to be partially applied. It is a conservative extension to the language to add these definitions, but they were omitted to keep the core language small.

With this constructor notation the syntax gets extended to:

$$
\begin{array}{rcll}
expr & ::= & \cdots & \text{the rules defined above} \\
 & | & C\ expr_1 \ldots expr_n & (\text{Constructor, } n \geq 0)
\end{array}
$$

$$
\begin{array}{rcll}
pat & ::= & _ & (\text{Matches everything}) \\
 & | & C\ pat_1 \ldots pat_n & (\text{Matches constructor } C \text{ and tests all arguments})
\end{array}
$$

$$
\begin{array}{rcl}
varpat & ::= & var\,@\,_ \\
 & | & var\,@\,C\ varpat_1 \ldots varpat_n
\end{array}
$$

As can be seen, the difference between *pat* and *varpat* only lies in the additional variables, which are included in *varpat*. These variables are used to bind the match of the corresponding subpattern. An example for this will be given in a moment.

We further define a relation $p_1 \sqsubseteq p_2$ operating on patterns and determining whether pattern p_1 is included (i.e. more specific) in p_2. It is obvious, that all patterns are more specific than _. Also note, that patterns are only comparable when using the same constructors. Hence A and B are incomparable as are $A\ _\ C$ and $A\ _\ D$. This relation can be defined via the set of constructors a pattern represents. So if $[\![p]\!]$ denotes the set of all constructors which are represented by p, we define $p_1 \sqsubseteq p_2 \iff [\![p_1]\!] \subseteq [\![p_2]\!]$.

From this follows that there are patterns, which do use the same constructors (in the pattern) but are also incomparable. This happens when the intersection of the sets of constructors is not empty but neither one is a subset of the other. An example is $p_1 = A\ _\ B$ and $p_2 = A\ B\ _$, where $A\ B\ B$ is represented by both, but $A\ C\ B$ only by p_1 and $A\ B\ C$ only by p_2.

Using this basic syntax it is already possible to write normal functional programs as all the

important parts are in place. So for example one can implement the fibonacci function[1]:

```
letrec fib = \x -> x | Int 0 -> Int 0
                     | Int 1 -> Int 1
                     | Int _ -> fib (x - Int 1) + fib (x - Int 2)
in fib (Int 20)
```

This example also shows the usage of patterns for the match statement: A pattern is a constructor or the underscore, where constructors only match themselves and underscore matches everything. This applies recursively to all the arguments of the constructor pattern.

The variables which are shown in the syntax can be used to reference the matched part of the pattern. As a showcase, the fibonacci function is modified to use variables:

```
letrec fib = \x -> x | Int (y @ 0) -> Int y
                     | Int (y @ 1) -> Int y
                     | y @ (Int _) -> fib (y - Int 1) + fib (y - Int 2)
in fib (Int 20)
```

It is obvious that when matching against a single variable, the outer-most variable in the pattern is equal to the matched one, as can be seen in the last case of this example.

3.1.3. Processes

The basic approach to concurrency in Grips are multiple processes running in parallel while the processes themselves are of a sequential nature. These processes only communicate with each other using messages and do not share variables besides some read-only environment. This kind of concurrency was first described by Hoare in [Hoa78] for his first revision of the language CSP (*Concurrent Sequential Processes*) and later revised into another language with the same name (see [Ros98] for a discussion on the concepts and history).

The general description of a process in CSP is that a process P of the form $a \to P'$ waits until it receives an a after which it behaves like the process P'. This is now mapped to lambda-expressions in Grips: The expression $\lambda a.\, t$ is a process which waits until it receives the message a. When this has happened, it behaves like t. The syntactic element for sending messages is the function application. For the ease of programming and to comply with the mathematical foundation of a "function", sending messages *does not* alter the receiving process, but creates a new state of this process in which the message has been already received. Now the new state and the original process co-exist. Altering the receiving process itself would lead to a situation, where multiple uses of a process would yield different

[1] Given a function "-" working on integers has already been defined.

results even with the same arguments. This violates one major mathematical assumption on functions.

To illustrate this concept, we use the following example:

```
let f = \x -> \y -> skip
in let x = f A
```

Here f is a *handle* to a process which waits for two messages. After receiving them it behaves like the empty process. Thus the application $f\ A$ sends a message to the process f is pointing to, and hence x now is a handle to a process which waits for one more message before behaving like the empty process.

This notion of handles is not restricted to lambda expressions, but in fact every variable name is a handle to some process. In case this process is empty, a lambda expression or a constructor, it is fully evaluated and may receive messages, in all other cases it is subject to evaluation by the rules laid out in Section 3.2.

This behavior of variables being labels can be found in other languages, too. An example is C, where a variable is a label for a memory cell and assigning a value to this variable stores this value in the referred memory cell. And also similar to these languages, the differentiation between a handle and the process it is pointing to is not always made explicit. So we will say "f is a process" instead of "f is a handle which points to a process", quite as one uses to say "x is 1" instead of "x points to a memory cell which contains the integer 1".

The presented simple theory of processes can be enriched by choice, which means that a process behaves differently depending on the message it receives. In CSP it is written $a \to P_1 \mid b \to P_2$, stating that the process behaves like P_1 if an a is received and as P_2 if a b is sent[2]. The counterpart in Grips is the match-statement

$$\begin{pmatrix} ?r \mid_{m_1} y_1@p_1 \to t_1 \\ \vdots \\ \mid_{m_n} y_n@p_n \to t_n \end{pmatrix};\ t$$

Here, depending on which pattern p_i matches r, the corresponding branch t_i is chosen. If the last statement in t_i is a `skip`[3], then the evaluation continues after t_i with the *fall-through* part t. This is also the difference between the two no-op statements `end` and `skip`, as `end` ends the process when reached.

Using this notion of processes, it is possible to create functions which accept an infinite amount of messages. This can be implemented by a process that does a final recursive call.

[2]Sending messages different from a or b is not allowed and not handled explicitly.
[3]Note that a `skip` is sometimes added implicitly as described in Appendix A.2.

3.1.4. Futures

The Grips constructs presented so far only allow messages in one direction: It is not possible for a process to send a message back to the process it received a message from. This ability is added now by the construct of *futures*. These are variables[4], which are marked as being filled eventually with a value (refer to [Hal85] for an introduction). This means that upon the creation of a future one holds a handle to an eventually filled value and can pass this handle around while the value is computed in parallel. If one wants to use the value, the content of the handle is examined: Either it is filled already and one uses the contained value, or it is still empty and the current process blocks until the future is filled with a value.

Adding the concept of futures requires the following additions to the syntax:

$$\begin{array}{rll} expr ::= & \cdots & \text{the rules defined above} \\ | & var_1, var_2 \leftarrow expr_1\,;\ expr_2 & \text{(Future sending)} \\ | & !var & \text{(Wait)} \\ | & \texttt{bind}\, var = expr_1\,;\ expr_1 & \text{(Future binding)} \end{array}$$

Adding the `bind` construct additionally to the existing `let` and `letrec` is done to easily differentiate syntacticly between future bindings and variable creation. This differentiation is especially important for the type inference as presented in Chapter 4.

The two *var*s in the construct for sending futures are the name of the future and the process $expr_1$ becomes after sending the future, respectively. To illustrate this concept, we will show a short example:

```
letrec f = \x -> \y -> bind x = y
in r, f’ <- f
   f’ (Int 10)
```

Here, after the second line, r is an unbound future: It has been send to f but not yet been bound to any value. And f' is the process $\lambda\, y.\ \texttt{bind}\, x = y\,;\ \texttt{skip}$, which is the part of f after receiving x, but with x contained in the local environment of f'. And finally, after the last line, r is bound to Int 10.

Using these futures it is now possible to compute values concurrently. Hence we enhance our fibonacci example from above such that the `fib` function now does not evaluate to a value, but instead receives a future and fills this with the result:

[4] Which is equivalent to processes as outlined in the previous section.

```
letrec fib = \x -> \res -> x | Int 0 -> bind res = Int 0
                             | Int 1 -> bind res = Int 1
                             | Int _ -> fib_1 <- fib (x - Int 1)
                                        fib_2 <- fib (x - Int 2)
                                        bind res = (! fib_1) + (! fib_2)
in r <- fib 20
   -- do something completely unrelated here
   -- ...
   ! r
```

The call $r \leftarrow fib\ 20$ immediately returns, with the future r as the result. The result of $fib\ 20$ is computed in parallel and the current thread of execution does not care about it until the point where the value of r is retrieved.

A special case with respect to futures are constructors: They behave implicitly like a process which awaits a future that is filled with the constructor itself. Thereafter the constructor behaves like the empty process. Hence the type of a constructor A is $\&A \to \varepsilon$, where $\&A$ stands for "a future of type A."

When used in actual code, this looks like:

```
let x = A 2 in
y, x' <- x
-- now y is equal in behavior to x
x'' <- x' -- this is not allowed, as x' is the empty process
```

Please note that the concept of futures leaves the world of CSP, as CSP does not allow shared state between processes – and a future is such a shared state. It even is possible to send the same future to different processes. This can for example be achieved by sending a future to the identity function, where you gain an unbound future that can be passed on by normal function application to other processes. As a possible use-case, these different processes may be controlled by other messages to determine which of them actually binds the future (as multiple binding is not legal) and which of them just uses it. A larger commented example program is given in Fig. 3.1.

3.2. Operational Semantics

While the previous section dealt with the syntax of Grips, this section will specify the behavior of a Grips program. This will be done by giving a small-step operational semantics (cf. Section 2.2 for an introduction to this topic).

```
-- f receives a future
-- if the next message is
--     A: x is bound to True
--     B: the process evaluates to the content of x
--     everything else: the message is ignored
letrec f = \x -> \y -> y | A -> bind x = True
                         | B -> !x
                         | _ -> f x
in
-- the identity function
let id = \x -> x in
r <- id -- r is now an unbound future
let g = f r in
let h = f r in
-- by assigning to a variable the application is evaluated
-- in a separate process
let g' = g A in
h C B -- C is ignored
-- at this point the current process has evaluated to True
```

Figure 3.1.: Example, showing multiple future passing

Notation

We present the semantics in terms of SOS-style inference rules [Plo81], where each state is a set of processes and "$\longrightarrow \subseteq S \times S$" is the evaluation relation, whereby S denotes the set of states. Thus $t \longrightarrow t'$ expresses "the state t is evaluated to the state t'". These rules use certain meta-variables, which are:

- $\mathbb{E}$: the set of all expressions (defined in the previous sections)
- $\mathbb{V}$: the set of all variables (defined in in Sec. 3.1.1)
- $\mathbb{P}$: the set of all processes
- $x, y, r, f, g, h \in \mathbb{V}$: different variable names. r is used for futures, f, g and h for functions and processes, and x, y are used for variables without a certain specification.
- $e, t \in \mathbb{E}$: expressions
- $\top$: the value of an uninitialized future

Further, with $\hat{\sigma}(t)$ we denote a substitution which replaces all bound variables in t by fresh ones.

The current state of an evaluation of a Grips program consists of a set of processes. Each of the processes is evaluated separately and the order of evaluation of the processes is nondeterministic. Hence each of the following rules describes one step of evaluation of one particular process, though the evaluation might create new processes. All other processes not used in the rule are contained in the mapping $\Pi \subseteq \mathbb{V} \times \mathbb{P}$, which maps handles to their respective process. Thus the rule

$$\frac{\textit{premises}}{\Pi \mid h\colon t \longrightarrow \Pi \mid h\colon t' \mid h'\colon t''}$$

has to be read as: "Given certain premises, any process t is evaluated to t' and a new process t'' is created under a (new)[5] handle h'."

In Section 3.1.3 it has already been stated that a process can either be fully evaluated or be subject to evaluation. Whenever a rule uses another process, it is assumed that this process is fully evaluated, which is the case, if it is either empty, a lambda expression or a constructor. If this does not hold (i.e. the process used is not fully evaluated), the rule cannot be applied.

If there is no rule which can be applied for any handle, the program is stuck, which indicates an error.

Semantic rules

The difference between `end` and `skip` only shows in match statements. Hence if the whole process only contains an `end`, it behaves the same as the empty process.

$$\frac{}{\Pi \mid h\colon \texttt{end} \longrightarrow \Pi \mid h\colon \texttt{skip}} \quad \text{(E-End)}$$

$$\frac{\Pi(f) = t_f \qquad \Pi(f) \neq \top}{\Pi \mid h\colon f \longrightarrow \Pi \mid h\colon t_f} \quad \text{(E-SingleHandle)}$$

$$\frac{\text{fresh}\, x_i'}{\Pi \mid h\colon C\ x_1 \ldots e_i \ldots e_n \longrightarrow \Pi \mid h\colon C\ x_1 \ldots x_i' \ldots e_n \mid x_i'\colon e_i} \quad \text{(E-Construct)}$$

The rule E-Construct states that expressions in arguments to a constructor are evaluated to handles, until the constructor only holds handles.

The next rule describes the behavior for dereferencing futures: Given that the variable

[5] Depends on the context.

dereferenced is filled with a value (keep in mind that & expresses "future" which differentiates it from normal processes), this value is copied into a fresh process, whose handle the current process evaluates to. A rule for an unfilled future (i.e. the content of the future is $\top$) does not exist – as long as the future is empty no rule can be applied to it.

$$\frac{\Pi(r) = \&e \qquad \text{fresh}\, h'}{\Pi \mid h\colon !r \longrightarrow \Pi \mid h\colon h' \mid h'\colon e} \quad \text{(E-WaitRead)}$$

The following rules for the different kinds of binding are similar to those of other functional languages. Due to the global namespace of handles, it must be guaranteed that each new variable has a name different from existing ones. Otherwise the old process of the variable would be overwritten by the new definition. To avoid these clashes of variable names, the variable being bound to is replaced by a fresh one. The only difference for the recursive case is that this substitution is also done for the expression which is bound, because the newly created variable might be used there.

$$\frac{\text{fresh}\, x'}{\Pi \mid h\colon \texttt{let}\, x = e\, \texttt{in}\, t \longrightarrow \Pi \mid h\colon t[x'/x] \mid x'\colon e} \quad \text{(E-Let)}$$

$$\frac{\text{fresh}\, f'}{\Pi \mid h\colon \texttt{letrec}\, f = t_f\, \texttt{in}\, t \longrightarrow \Pi \mid h\colon t[f'/f] \mid f'\colon t_f[f'/f]} \quad \text{(E-LetRec)}$$

When it comes to future binding, this renaming is not possible, as the variable being bound to, has to match the name of the future which has been sent. The reason is the reliance on names to find the content of the handle in the global process space Π: If a fresh name would be used, the connection to the received future would be lost and hence the sending process would never receive the content.

Furthermore, due to the & a future is syntacticly different from a normal process and hence these rules would not match the expression put into the future. But this prevents concurrent evaluation of the future and therefore instead of saving the expression itself, another handle is created which holds the expression and can evaluate it in parallel. The future only holds this handle.

$$\frac{\Pi(r) = \top \qquad \text{fresh}\, r'}{\Pi \mid h\colon \texttt{bind}\, r = e\,;\; t \longrightarrow \Pi \mid h\colon t \mid r\colon \&r' \mid r'\colon e} \quad \text{(E-Bind)}$$

The following two rules deal with function application: They ensure, that function and arguments are put into handles. This way, as all processes are evaluated in parallel

independent of their usage, the argument evaluation is strict. Further the evaluation of the arguments is stated to happen from left to right.

$$\frac{\text{fresh}\, f}{\Pi \mid h\colon e_1\ e_2 \longrightarrow \Pi \mid h\colon f\ e_2 \mid f\colon e_1} \quad \text{(E-ApplyStep)}$$

$$\frac{\text{fresh}\, g}{\Pi \mid h\colon f\ e \longrightarrow \Pi \mid h\colon f\ g \mid g\colon e} \quad \text{(E-ApplyStepArgument)}$$

After this replacement has been done, the body of the called function is copied into the current process. With the use of the $\hat{\sigma}$ substitution, all bound variables in the body get replaced by fresh names (e.g. in $\texttt{let}\, x = y\, \texttt{in}\, x$, the variable x gets substituted, but not y). This avoids name clashes as variables in the body may have the same name as variables in the context of the current process.

$$\frac{\Pi(f) = \lambda\, x.\, t \qquad g \in \text{dom}(\Pi)}{\Pi \mid h\colon f\ g \longrightarrow \Pi \mid h\colon \hat{\sigma}(t)[g/x]} \quad \text{(E-Apply)}$$

It has been described in the section introducing futures (Sec. 3.1.4), that futures can be sent to constructors. This is a special case of application and the next rule implements that part.

$$\frac{\Pi(f) = C\ x_1 \ldots x_n \qquad \Pi(r) = \top}{\Pi \mid h\colon f\ r \longrightarrow \Pi \mid h\colon \texttt{skip} \mid r\colon \&C\ x_1 \ldots x_n} \quad \text{(E-ApplyConst)}$$

The previous rule already showed that sending futures is a normal application with the additional constraint, that the message is $\top$. Consequently the semantics of the future-sending-expression are exactly that: Creating a new handle with the $\top$-process and apply it.

$$\frac{\text{fresh}\, r' \qquad \text{fresh}\, y'}{\Pi \mid h\colon r, y \leftarrow f\ ;\ t \longrightarrow \Pi \mid h\colon t[r'/r, y'/y] \mid r'\colon \top \mid y'\colon f r'} \quad \text{(E-SendFuture)}$$

Now follows the slightly more complicated construct of the match: The core construct only matches against a future[6]. Further it contains a fall-through expression, which is only reached by those branches ending in `skip`. This has been described in Section 3.1.3 and will be handled by the function $s\colon \mathbb{E} \times \mathbb{E} \to \mathbb{E}$, whose rules are given in detail in Fig. 3.3. This function receives a branch and the fall-through expression and creates the final expression to evaluate by replacing a final occurring `skip` by the fall-through expression.

[6] This is the difference between $?r \mid \ldots$ and $r \mid \ldots$, the latter being just syntactic sugar. The exact rules are laid out in appendix A.2 on page 54.

$$\texttt{createprocs}(y@(x@A\ z@_), A\ B) = (\{x'\colon A, z'\colon B, y'\colon A\ B\}, [z'/z, y'/y, x'/x])$$
$$\texttt{createprocs}(y@(x@A\ z@B), A\ B) = (\{x'\colon A, z'\colon B, y'\colon A\ B\}, [z'/z, y'/y, x'/x])$$
$$\texttt{createprocs}(y@_, A\ (B\ C)\ C) = (\{y'\colon A\ (B\ C)\ C\}, [y'/y])$$
$$\texttt{createprocs}(y@(x@A\ u@_\ v@C), A\ (B\ C)\ C) = (\{y'\colon A\ (B\ C)\ C, x'\colon A, u'\colon B\ C, v'\colon C\}, [y'/y, x'/x, u'/u, v'/v])$$

Figure 3.2.: Examples of `createprocs` uses

The match statement is handled by two different rules: The first one handles the case, where a match is found, the second one handles the opposite case of failure. For determining whether a certain expression matches a pattern, these rules use the function `matches`, which takes care of extracting the current pattern out of the expression (it may be needed to resolve multiple indirections through handles) and comparing it against the given var-pattern.

$$\frac{\Pi(r) = \&e \qquad \texttt{matches}(e, p_1) \qquad \hat{\Pi}, \sigma = \texttt{createprocs}(y_1@p_1, e)}{\Pi \mid h\colon \begin{pmatrix} ?r \mid_{m_1} y_1@p_1 \to t_1 \\ \vdots \\ \mid_{m_n} y_n@p_n \to t_n \end{pmatrix};\ t \longrightarrow \Pi \mid h\colon s(\sigma(t_1); \hat{\sigma}(t)) \mid \hat{\Pi}} \quad \text{(E-MatchHandleMatch)}$$

The function `createprocs` used in the previous rule creates a fresh handle for each variable in the matched var-pattern, which points to the matched sub-pattern (creating the set of handles $\hat{\Pi}$). It further returns a substitution σ from the variables of the var-pattern to their respective handle in $\hat{\Pi}$. Some examples for this function can be found in Fig. 3.2. This substitution then is applied to the branch statement. Thereby the values of the matched expressions are brought into scope. Additionally the substitution $\hat{\sigma}$ is applied to the fall-through statement to avoid name-clashes with variables introduced in the branch.

$$
\begin{aligned}
s(\texttt{end}, t) &\equiv \texttt{end} \\
s(\texttt{skip}, t) &\equiv t \\
s(t, \texttt{skip}) &\equiv t \\
s(C\ x_1 \dots x_n, t) &\equiv C\ x_1 \dots x_n \\
s(!x, t) &\equiv !x \\
s(e_1\ e_2, t) &\equiv e_1\ e_2 \\
s(\texttt{bind}\, x = e\,;\ t_1, t_2) &\equiv \texttt{bind}\, x = e\,;\ s(t_1, t_2) \\
s(\texttt{let}\, f = e\, \texttt{in}\, t_1, t_2) &\equiv \texttt{let}\, f = e\, \texttt{in}\, s(t_1, t_2) \\
s(\texttt{letrec}\, f = e\, \texttt{in}\, t_1, t_2) &\equiv \texttt{letrec}\, f = e\, \texttt{in}\, s(t_1, t_2) \\
s(r, f' \leftarrow f\,;\ t_1, t_2) &\equiv r, f' \leftarrow f\,;\ s(t_1, t_2) \\
s(\lambda\, x.\, t_1, t_2) &\equiv \lambda\, x'.\, s(t_1[x'/x], t_2) \quad \text{fresh}\, x' \\
s\left[\begin{pmatrix} ?r \mid_{m_1} y_1@p_1 \to t_1 \\ \vdots \\ \mid_{m_n} y_n@p_n \to t_n \end{pmatrix};\ t_m, t\right] &\equiv \begin{pmatrix} ?r \mid_{m_1} y_1@p_1 \to t_1 \\ \vdots \\ \mid_{m_n} y_n@p_n \to t_n \end{pmatrix};\ s(t_m, t)
\end{aligned}
$$

Figure 3.3.: Transformation rules

The following failure rule is simpler, as the non-matching case is simply discarded.

$$
\frac{\Pi(r) = \&e \qquad !\texttt{matches}(e, p_1)}{\begin{array}{l} \Pi \mid h\colon \begin{pmatrix} ?r \mid_{m_1} y_1@p_1 \to t_1 \\ \quad \mid_{m_2} y_2@p_2 \to t_2 \\ \vdots \\ \quad \mid_{m_n} y_n@p_n \to t_n \end{pmatrix};\ t \\ \longrightarrow \Pi \mid h\colon \begin{pmatrix} ?r \mid_{m_2} y_2@p_2 \to t_2 \\ \vdots \\ \quad \mid_{m_n} y_n@p_n \to t_n \end{pmatrix};\ t \end{array}} \quad \text{(E-MatchHandleFailure)}
$$

4. Development of the type system

In this chapter the development of the type system will be presented and discussed. It starts with defining the set of types used in that system. It continues with giving the description of the construction of an automaton for the control flow in a function and then describe the semantics for collecting the constructors which are used for any variable. Finally we are building a system of implications which also model the flow of control, but now in a more specified and finer way by tracking each occurrence of a constructor.

These four different chapters all have their own notation as they try to achieve different goals by different means. Therefore it did not seem wise to give an overview of all notions used in the upcoming sections. Instead we solely state those sets and functions utilized in all sections.

- $\mathbb{E}$: the set of expressions
- $\mathbb{V} \subset \mathbb{E}$: the set of variables
- $\mathbb{T}$: the set of types
- $\mathbb{P}$: the set of patterns as defined by Fig. 4.1. The definitions from Section 3.1.2 apply.
- $\mathbb{M}$: the set of match branches
- $\mathcal{M}\colon \mathbb{M} \to \mathbb{P}$: a relation mapping labels of branches of match statements to patterns. This is used to store the complete pattern a particular branch matches.

The first four sets should be self-explanatory and $\mathbb{E}$ and $\mathbb{V}$ have moreover been defined while introducing Grips. They are the very same sets as given in Chapter 3 . In $\mathbb{M}$ each branch of a match statement is an element, and such a branch is represented as a label. These labels can be seen in the syntax as given in the previous chapter:

$$?r \mid_m _ \to t$$

Here the label m represents the current branch.

$$
\begin{aligned}
pat ::= \; & _ \\
| \; & cons\; pat_1 \dots pat_n \quad n \geq 0 \\
\\
cons ::= \; & \texttt{[A - Z][a - zA - Z0 - 9]*}
\end{aligned}
$$

Figure 4.1.: Pattern definition

4.1. Description of the types

In difference to the well-known Hindley-Milner typing system [Hin69][Mil78] and the Damas-Milner type inference algorithm $\mathcal{W}$ [DM82], the typing system presented in this thesis allows a function to have different types for the different branches of execution – given the branching happens on the message received last. From this follows, that the correct typing of a branch depends on the type of the last message.

This needs to be reflected in the way types are to be defined in this system. Another point is the content of a type: What exactly does a certain type τ stand for?

The latter point can be answered by seeing the only way data can be defined in Grips (cf. Sec. 3.1.2) – by constructors. Advanced concepts like type-classes [WB89] are not part of this language, which allows to have sets of constructors as a basic type. For the matter of typing, also futures (&) and functions ($\rightarrow$) are seen as constructors, though they do not share the other properties of syntactic constructors as outlined in Sec. 3.1.2, which first and foremost implies, that the inference rules of constructors are not applicable to them. A further special "constructor" is the *unit constructor* (). It is used only for controlling the flow of types insides matches as will be detailed later on in this section.

These points can be summed up by the following definition of a type $\tau \in \mathbb{T}$:

$$
\begin{aligned}
\tau ::= \; & \{t_1, \dots, t_n\} & \\
t ::= \; & cons\; \tau_1 \dots \tau_n \quad (n \geq 0) & \text{(Constructor)} \\
| \; & \tau_1 \xrightarrow{P \in \mathbb{P}} \tau_2 & \text{(Function Type)} \\
| \; & \&\tau & \text{(Future)} \\
| \; & () & \text{(Unit)}
\end{aligned}
$$

The P in the definition of the function type fulfills the demand for different branch-types depending on the sent message as described in the beginning of this section: It is a pattern, the last message (of type τ_1) has to match, so that the function returns a type τ_2. So if a

function has the type

$$\{\{A, B, C\} \xrightarrow{A} \tau_1, \{A, B, C\} \xrightarrow{_} \tau_2\}$$

it results in type τ_1 if a message A is sent, and in τ_2 for all other cases.

As the reader might notice, this additional pattern might be redundant, as it could be expressed directly in the type of the message – thus yielding

$$\{\{A\} \to \tau_1, \{B, C\} \to \tau_2\}$$

for the previous example. But this is in fact not possible, as the type of the first message is equal to the variable representing the message inside the function body. And this variable has to have the complete set of constructors it can represent in this function. So the additional pattern is indeed required to allow the wanted behavior of types in branches.

Representing the branching of types inside the function type is still not sufficient, as the branching happens at match-statements and it is thus necessary to be able to represent this behavior already at this point. Therefore the plain τ gets extended and is annotated with the pattern the last message must have matched. This leads to the definition of the *constrained type* set $\bar{\tau}$:

$$\bar{\tau} ::= \langle \tau_1|_{P_1}, \ldots, \tau_n|_{P_n} \rangle$$

To enhance readability and reduce irritations the excessive usage of set braces might entail, angle brackets were preferred. Nonetheless $\bar{\tau}$ is still a set and hence the ordering of types is of no importance.

Throughout this thesis, a $\bar{\tau}$ in most cases is of the form $\langle \tau|_ \rangle$ and thus this extra abstraction may be ignored by the reader up to the point where it will be used in more complex variants.

A last extension to this system of types is induced by the existence of the fall-through expression inside a match-statement. For this reason it is again necessary to express the difference in types between a branch, which further continues to the fall-through part, and another branch which does not. Additionally, in matters of typing, the type of a branch *continues* at the fall-through point. So, given the following example,

```
x | A -> \y -> skip
  | B -> C
  D
```

the first branch falls through and thus gets the type[1] $\langle \{_ \xrightarrow{_} \{D\}\}|_A \rangle$. In contrast, the second branch ends directly at the C and hence gets the type $\langle \{C\}|_B \rangle$.

[1]The property of constructors to implicitly await a future is omitted here for brevity.

We achieve this behavior by adding another type to each statement leading to the complete type expression of a syntactic construct:

$$\bar{\tau}_i \rhd \tau_o$$

$\bar{\tau}_i$ is a constrained type as defined above, and τ_o is a normal type. But while $\bar{\tau}_i$ represents the type of the whole expression, τ_o states whether this expression qualifies for fall-through in a match: If it is (), fall-through does not happen, else it is the type of the last sub-expression in the current expression. Mostly for historical reasons, they are referred to as *input* ($\bar{\tau}_i$) and *output* (τ_o) types, though τ_o is indeed the type that “comes out” of a branch in a match.

As types are sets and sets allow subset relations, the type system does allow subtyping [Rey98][Pie02]. Hence, if $\tau_1 \subseteq \tau_2$ holds, one can use τ_1 in places where τ_2 is expected. This must not be confused with polymorphism, which requires that expressions can have an “infinite family of types” [Rey98, p. 379], which is normally implemented by allowing types to contain type variables. These can then be instantiated with different types itself. As there is only a finite number of subsets of a type, this type system is monomorphic according to Reynolds.

On the other hand, Cardelli defines subtyping as another form of polymorphism [CW85] called *inclusion polymorphism*. Together with the *parametric polymorphism*, that fits the definition of Reynolds, they both are part of *universal polymorphism*. Therefore we refrain from giving an explicit statement to whether the type system of this thesis is mono- or polymorphic.

Also it must be taken care to not create infinite types like $\tau = \{C\ \tau\}$. This is currently not supported by the presented system.

4.2. Automata construction

In this section, an algorithm is presented that creates an automaton which will correctly model the branching behavior of a function. Hence it is possible to see the flow of control induced by messages.

This automaton uses the process-based notion we introduced in Section 3.1.3: It describes what a certain process does after receiving a certain message. Therefore the nodes in this automaton represent a process which awaits a message (i.e. a lambda expression) and the edges are labeled with the pattern of the received message. An example is presented in Fig. 4.3. The program which would result in such an automaton could be for example the one shown in Fig. 4.2.

Note that such an automaton is not unique, i.e. a family of programs map to the same

```
let f =
    let g = \y -> y | Int 2 -> return False
    in
    \x -> x | A C _ -> g
            | C -> g
            | _ -> return True
```

Figure 4.2.: Example program

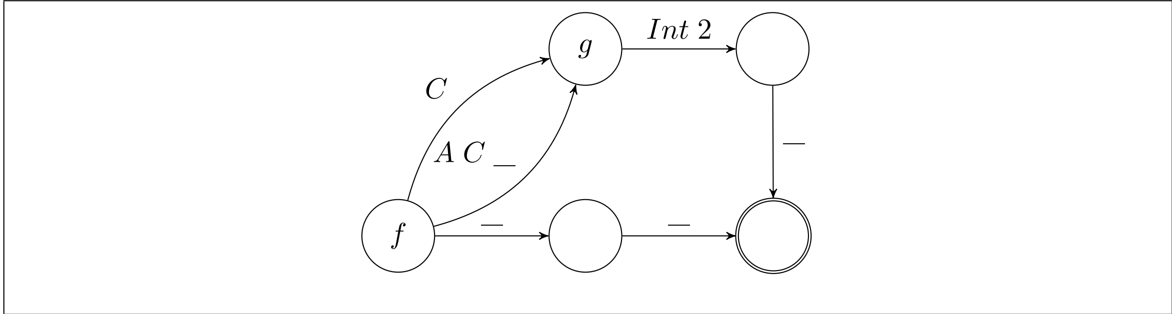


Figure 4.3.: Example automaton

automaton.

4.2.1. Notation

For each global function f, we create an automaton

$$A_f = (q_0, Q, V, F, \Sigma, \delta, \Phi, \Gamma)$$

where Q, $q_0 \in Q$, and $F \subseteq Q$ have their common meaning (refer for example to [HMU00] for an introduction in automata theory) of set of states, starting state, and set of final states, respectively. The other parts are:

- $\Sigma \subseteq \mathbb{P}$: the alphabet, being the set of all possible patterns.
- V: a set of virtual nodes, which are used to model calls to external functions. Virtual nodes are represented as $[g\ C_1 \ldots C_n]$, where $g \in \mathbb{V}$ is the function name and $C_1 \ldots C_n$ ($C_i \in \mathbb{P}$) are the arguments. Given a virtual node v, these parts are accessed via `name` v and `args` v, respectively. With $[[g\ C_1 \ldots C_n] + C_m \ldots C_k]$ a new virtual node $[g\ C_1 \ldots C_n\, C_m \ldots C_k]$ is created. Note: $Q \cap V = \emptyset$.
- $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times (Q \cup V)$: the transition relation. Note that it is defined differently from the δ found in common automata definitions.

- $\Phi \subseteq \mathbb{V} \times Q$: a relation mapping function names to a state in the automaton
- $\Gamma \subseteq \mathbb{V} \times 2^V$: a relation mapping variable names to a subset of V. This mapping is used to hold the set of virtual nodes the application of a variable might produce. This is needed, when a variable can point to different functions depending on runtime behavior.

Additionally, the following parts make use of certain variables. These are

- $S \subseteq Q$: a set of temporary nodes. These nodes should not be reachable in the final automaton. In this thesis, they are represented by $*_e$ for any e.
- ℓ: the current label. See section 4.2.2 for details on the functionality.
- $\alpha \subseteq \mathbb{V}$: the set of current variables, which still allow branching
- $\beta \subseteq \mathbb{V} \times \mathbb{V}$: a mapping from variables to variables. This tracks sending futures, i.e. if x is a future sent to y, then $(x, y) \in \beta$.
- $\hat{q} \in Q$: current state
- $\omega \in Q$: holds the state which is jumped to when the command is evaluated. This is used in match-statements.

For implementation reasons, in the following algorithms those variables (except S) are used as if they were the top of stacks. This means, that for example ℓ is just the short notation for `top` $\bar{\ell}$, where $\bar{\ell}$ is a stack of labels and `top` s returns the top-most element of any stack s.

4.2.2. Edge refinement

We outlined in the introduction to this section, that the reason of the automaton is to give detailed information about the effect of different constructors in an argument. Therefore it is essential to gather the correct information. This especially includes nested matches, where the pattern of the first match gets more definite with each nesting step. Such an example is given in Fig. 4.4 with the respective automaton in Fig. 4.5.

For this task we define a datatype to represent the label of an edge, which can be refined over multiple steps. This datastructure `Label` is created by the function `new-label` (see Algorithm 4.1) and has two methods: `lab` (see Algorithm 4.2) and `refine` (see Algorithm 4.3), where `lab` returns the current string representation of the label, and `refine` returns a new refined `Label`.

```
f = \x -> x | A y@_ -> y | B -> \z -> \z' -> end
                         | C -> \z -> z | Y -> end
                         | _ -> end
              | _ -> return False
```

Figure 4.4.: Code example of nested refinement

Input: variable name x
-- *var: variable this pattern is used for*
-- *pat: the pattern string*
-- *subs*: $\mathbb{V} \rightarrow$ `Label`: *holds the sub-patterns*
$\ell = \{var : x, pat : _, subs : \emptyset\}$
return ℓ

Algorithm 4.1: Creating a new label

Input: label ℓ
Output: transition string
if $\ell = \varepsilon$ **then**
  return ε
else
  if $\ell.pat = _$ **then**
    return $_$
  else
    $pat := \ell.pat$
    for each variable v in pat **do**
      replace v by `lab` $\ell.subs(v)$ in pat
    done
    return pat
  fi
fi

Algorithm 4.2: Label printing

```
Input: label ℓ
Input: variable x to refine
Input: refinement pattern pat
Output: refined pattern
  copy-ℓ := copy ℓ
  if pat ≠ _ then -- refine by _ useless
    if x ≠ ℓ.var then -- Refinement of a subpattern
      copy-ℓ.subs[x ↦ refine ℓ.subs(x) x pat]
    else if constructor and arity of pat and ℓ.pat differ then
      fail "refinement mismatch"
    else if ℓ.pat = _ then
      copy-ℓ.pat = pat
      for each variable v in pat do
        copy-ℓ.subs(v) = new-label v
      done
    else
      for each argument a of pat do
        v := variable corresponding to a
        copy-ℓ.subs[v ↦ refine ℓ.subs(v) v a]
      done
    fi
  fi
  return copy-ℓ
```

Algorithm 4.3: Label refinement

```
push ā₁ x₁
...
push āₙ xₙ
𝒜(t)
pop ā₁
...
pop āₙ
```

Algorithm 4.4: Explicit behavior of the preserving call

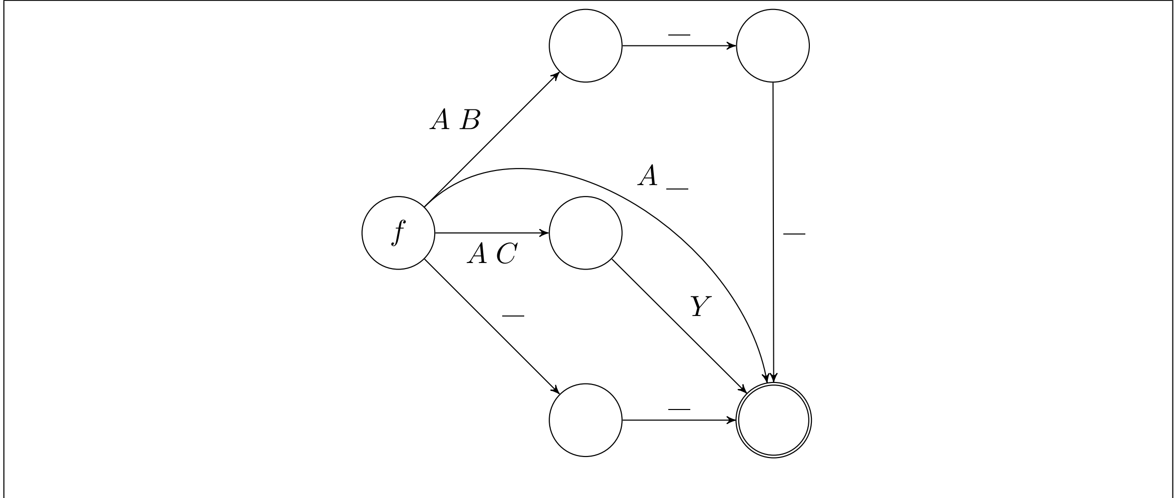


Figure 4.5.: Nested automaton

4.2.3. The algorithm $\mathcal{A}$

The algorithm to create the automaton for the function f is presented as an inductive definition. The parts of each construct of the language are given in an imperative style for better readability, but could be represented in normal functional style quite easily. Using a functional style would amongst others imply that the parts of the automaton A_f, which are considered global state, have to be passed around explicitly.

Also by using the notation $\mathcal{A}(t)\langle a_1 = x_1, \ldots, a_n = x_n\rangle$ we want to express that the state of the variables a_1 to a_n is preserved and is set to their respective values x_1 to x_n for the application of $\mathcal{A}$ on t. With introducing the notation in Section 4.2.1 we also mentioned that variables are just implicit references to the top of stacks. These stacks can now be used for state-preservation and hence $\mathcal{A}(t)\langle a_1 = x_1, \ldots, a_n = x_n\rangle$ can be seen as syntactic sugar for explicit stack operations, which are shown in Algorithm 4.4.

Moreover we assume that the compiler changed those applications, where the first part is not just a variable, to function applications inside a `let`-block. So $(!x)\ A$ would be changed to $\texttt{let}\, x' = !x\, \texttt{in}\, x'\ A$ or $(\lambda\, x.\, f\ x\ 2)\ C$ is transformed to $\texttt{let}\, x' = \lambda\, x.\, f\ x\ 2\, \texttt{in}\, x'\ C$.

Initialization

For each function where a complete and independent automaton should be created, the different parts of the automaton need to be initialized. Therefore everything is set to the empty set except:

- $Q := \{q_F\}$
- $F := \{q_F\}$

- $\omega := q_F$

This adds a final state q_F to the automaton. This final state is also the only one, i.e. F will not hold any other states throughout this algorithm. The use of ω will be explained in the coming section dealing with the `skip` statement.

The function is then just handled as a normal $\texttt{letrec}\, f = \lambda\, x.\, t$ by this algorithm.

Lambda-Expression

A lambda expression $\lambda\, y.\, t$ creates a new node and adds a transition from the current node to the newly created one, using the current label as the pattern on this transition:

$$Q := Q \cup \{q_y\}$$
$$\delta := \delta \cup \{(\hat{q}, \texttt{lab}\, \ell, q_y)\}$$
$$\mathcal{A}(t)\langle \ell = \texttt{new-label}\, y, \alpha = \{y\}, \hat{q} = q_y \rangle$$

Skip

The `skip` statement is an implicit statement for all statements which allow sequencing or scoping (these are all assignments, the match construct and the sending of futures). Hence it needs to be differentiated between the positions it can occur in: In most cases it signals the end of process, but while in a match the computation continues at the fall-through expression of the match.

This is the reason the variable ω has been introduced: It holds the node in the automaton a `skip` is jumping to – it is q_F for all cases except in matches where it is set to the node of the fall-through expression beforehand:

$$\delta := \delta \cup \{(\hat{q}, \texttt{lab}\, \ell, \omega)\}$$

End

The `end`-statement is similar to the `skip`, but *always* ends the process:

$$\delta := \delta \cup \{(\hat{q}, \texttt{lab}\, \ell, q_F)\}$$

Constructors

As pointed out earlier in Section 3.1.4, constructors implicitly receive a future which is filled with their own content and then are finished. This is reflected by their behavior in the

automaton, which adds another node that then goes with an _-transition into the final node:

$$Q := Q \cup \{q_C\}$$
$$\delta := \delta \cup \{(\hat{q}, \texttt{lab}\,\ell, q_C), (q_C, _, q_F)\}$$

Assignment

Assignments can be of three different kinds:

- *function definition*: $\texttt{letrec}\,x = e\,\texttt{in}\,t$
- *variable binding*: $\texttt{let}\,x = e\,\texttt{in}\,t$
- *future binding*: $\texttt{bind}\,x = e\,;\ t$

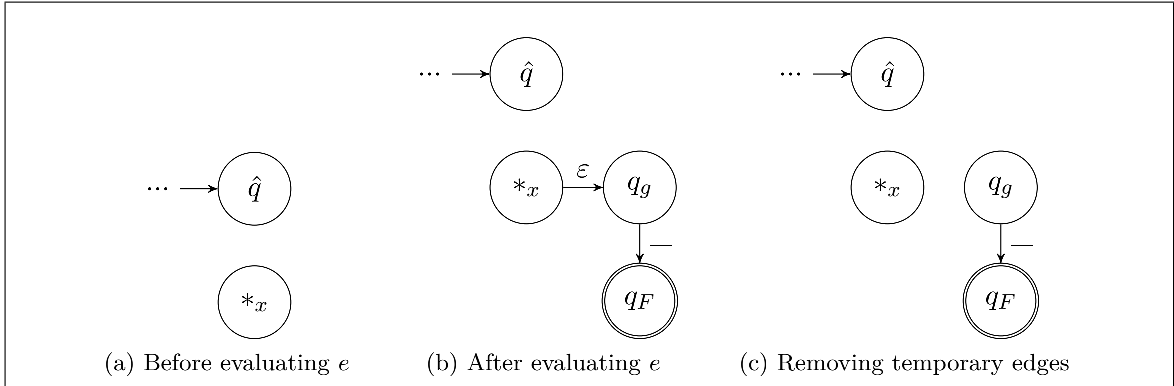
These three forms are not distinguished for this algorithm, though, as their semantic differences are without impact on the automaton:

-- *Evalute the assigned expression, but start with a fresh temporary node.*
$Q := Q \cup \{*_x\}$
$\mathcal{A}(e)\langle \hat{q} = *_x, \alpha = \emptyset, \ell = \varepsilon, \omega = \omega\rangle$
-- *Collect all virtual nodes which are directly reachable from* $*_x$
$\Gamma := \Gamma[x \mapsto \{v \mid \exists l.\ (*_x, l, v) \in \delta \wedge v \in V\}]$
-- *Collect all normal nodes directly reachable. This marks* x *as a function.*
if $\exists q \in Q \setminus (F \cup S):\ (*_x, \varepsilon, q) \in \delta$ **then**
$\quad \Phi := \Phi[x \mapsto q]$
fi
-- *Remove temporary edges and evaluate the enclosed expression*
$\delta := \delta \setminus \{(*_x, \varepsilon, q) \mid \forall q \in Q \cup V\}$
$\mathcal{A}(t)$

In Fig. 4.6 you find an example automaton showing the different steps of the presented algorithm for the code snippet

```
let g = \y -> end -- '\y -> end' is represented as 'e' in the automaton
```

In this example, the automaton is assumed to be constructed up to the current state $\hat{q}$. After the current algorithm is finished, you will notice, that the newly created node q_g is not connected to the existing automaton by an incoming edge. This is the correct behavior,



(a) Before evaluating e (b) After evaluating e (c) Removing temporary edges

Figure 4.6.: Creating the automaton of an assignment

as there should be only an edge to the node, if the flow of control ever reaches it, i.e. the function g is called. But with the current code above, this has not happened yet. If it is used as a function at a later point in the code, an edge might be added from the calling site to q_g. The details for this are given in Section 4.2.4.

Sending futures

Sending a future $(r, f' \leftarrow f\ ;\ t)$ is basically of no influence for the automaton, except that the name of the future is recorded in β so that the name of the future can later be related to the original variable. This is needed to determine, whether the variable is the last one being received and thus allows full branching.

$$\beta := \beta[r \mapsto f]$$
$$\mathcal{A}(t)$$

Function calls

In the introduction to this section it has been mentioned, that this algorithm assumes, that function applications always start with a variable. This especially removes the need for a special handling of inlined lambda abstractions. Hence a function call can always be assumed to be of the form $g\, C_1 \ldots C_n$ for $n \geq 0$. Such a call creates a *virtual* node which symbolizes a transition to a state in another automaton. Such a virtual node does not have any outgoing edges, which makes it quite similar to the final node. While the outgoing edges of a node to nodes in Q must be unique with respect to the labels, a node may have outgoing edges *with the same label* to multiple virtual nodes.

These virtual nodes are then handled later on. See Section 4.2.4 for details.

if $g \in \text{dom}(\Gamma) \wedge \Gamma(g) \neq \emptyset$ **then** -- *g is a variable, mapping to different functions*
$V := V \cup \{[x + C_1 \ldots C_n] \mid x \in \Gamma(g)\}$
$\delta := \delta \cup \{(\hat{q}, \texttt{lab}\,\ell, [x + C_1 \ldots C_n]) \mid x \in \Gamma(g)\}$
else -- *normal call*
$V := V \cup \{[g\ C_1 \ldots C_n]\}$
$\delta := \delta \cup \{(\hat{q}, \texttt{lab}\,\ell, [g\ C_1 \ldots C_n])\}$
fi

Match

With function calls and lambda expressions, the match-statement is one of the core parts for the automaton construction: This construct allows branching in the automaton, but only on the last message. Additionally, it has the fall-through expression, which is evaluated after some branches, depending on the content of the branch, as was presented in Section 3.1.3.

These different fields make the algorithm for the match-statement the most complicated of the algorithms presented in this section. It is presented in Algorithm 4.5 on page 35.

We will show an example of the automata construction of a match-statement in Fig. 4.7. Therefore, assume the following code given:

```
let f = \x -> x | A -> skip
                | B -> \y -> skip
                | C -> end
                D
```

The Fig. 4.7 starts with showing the automaton before the match, but already with the temporary node $*_t$ for the fall-through statement added. In figures (b) to (d) the different branches are added to the automaton and it can be clearly seen, that the `skip`s end in $*_t$, while `end` directly adds a transition to the final state. Please also note the additional state q_y which is created for the lambda expression, and that the following `skip` behaves similar to the one in the first branch.

Following in figure (e), the partial automaton of the fall-through statement D is added. As can be observed, it is connected via an ε-transition which marks this transition as temporary. And finally, in the last figure (f), the edges to $*_t$ and the ε-transition are combined by moving the edges to the state the ε-transition points to. This leaves the temporary node unconnected to the remainder of the automaton and hence it is unimportant for the rest of the algorithm.

From this example the meaning behind the temporary nodes becomes evident: They "bind" these edges where – at the point of construction – it is not possible to know the

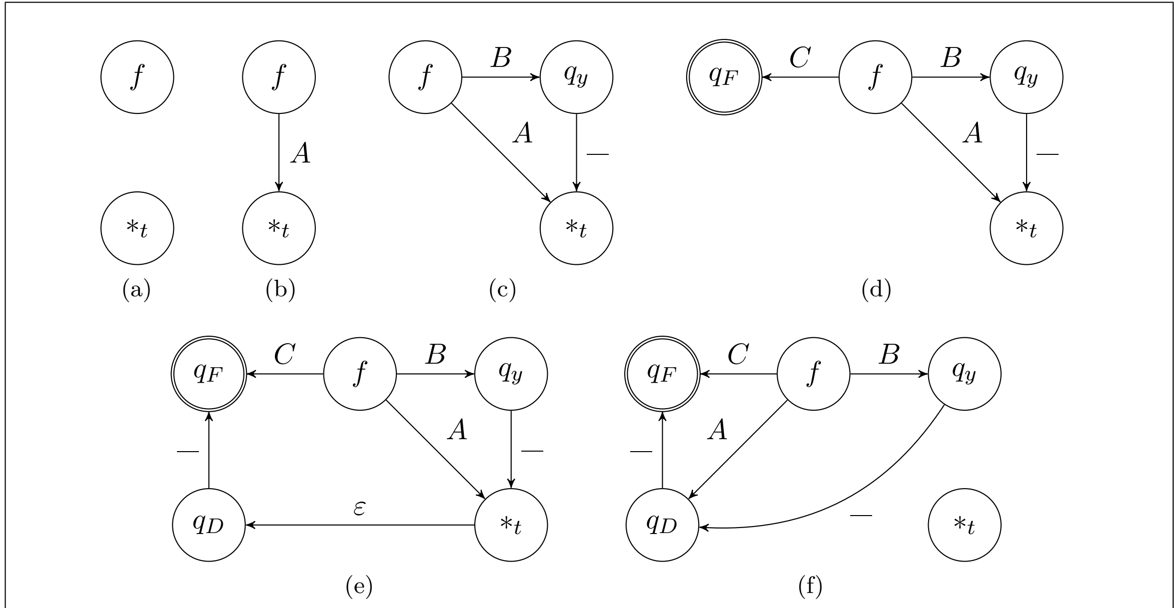


Figure 4.7.: Creating the automaton of a match

state, those transitions lead to. Only after this becomes known, the edges are moved to the appropriate target and the temporary node is obsolete.

4.2.4. Mapping virtual nodes

After generating an automaton using the described algorithm $\mathcal{A}$, this automaton might have virtual nodes. These virtual nodes are used to model calls to external functions. Unfortunately, virtual nodes are difficult to handle in the upcoming analyses as besides the type of arguments, nothing is known about the function called and it is not possible to see in which state in the automaton of the called function the call will end. Hence we want to reduce the number of possible virtual nodes by resolving virtual nodes leading to local functions, i.e. functions defined in the current function or the current function itself (= recursive calls).

While creating the automaton, no attention was paid to the locality of the called functions. Therefore, it is necessary to look at each virtual node and check whether the function called in the virtual node is in Φ. Remember that Φ maps from function names to states in Q. Hence, if a function $g \in \text{dom}(\Phi)$, it is known that execution can continue at the state $\Phi(g)$. So it remains to use each of the passed arguments to walk through the automaton and eventually replace the virtual node by the nodes which were reached during the transition. This is continued until a local fixpoint is reached, meaning that if during this transition a virtual node v is reached, v is going to be resolved immediately. If this is not possible, i.e. v

-- *Create the temporary node, which will be used for the fall-through expression.*

$Q := Q \cup \{*_t\}$

for $i := 1$ to n **do**

  -- *Find the message this future corresponds to.*

  $e := r$

  while $e \in \text{dom}(\beta)$ **do**

    $e := \beta(e)$

  done

  if $e \in \alpha$ **then** -- *branching on last message → refine*

    $\ell_m := \texttt{refine}\, \ell\ e\ p_i$

    $\alpha_m := \alpha \cup \texttt{vars}\ p_i \cup \{y_i\}$

    $\beta_m := \beta[y_i \mapsto e]$

    $\mathcal{M} := \mathcal{M}[m_i \mapsto \texttt{lab}\, \ell_m]$

  else

    $\ell_m := \ell$

    $\alpha_m := \emptyset$

    $\beta_m := \beta$

    $\mathcal{M} := \mathcal{M}[m_i \mapsto _]$

  fi

  -- *Evaluate the branch expression.*

  -- *With setting* $\omega = *_t$, *all* `skip` *expressions will jump to* $*_t$.

  $\mathcal{A}(t_i)\langle \ell = \ell_m, \alpha = \alpha_m, \beta = \beta_m, \omega = *_t \rangle$

done

-- *Evaluate the fall-through expression, starting with the temporary node.*

$\mathcal{A}(t)\langle \ell = \varepsilon, \alpha = \emptyset, \hat{q} = *_t \rangle$

-- *Resolve the* ε*-transitions from* $*_t$.

-- $q \xrightarrow{A} *_t \xrightarrow{\varepsilon} q'$ becomes $q \xrightarrow{A} q'$

$q_{next} := \{q \mid (*_t, \varepsilon, q) \in \delta\}$

$\delta := \delta \setminus \{(*_t, \varepsilon, q) \mid \forall q \in Q \cup V\}$

for all $(q, l, *_t) \in \delta$ **do**

  $\delta := \delta \setminus \{(q, l, *_t)\}$

  $\delta := \delta \cup \{(q, l, q') \mid q' \in q_{next}\}$

done

Algorithm 4.5: Match Rule

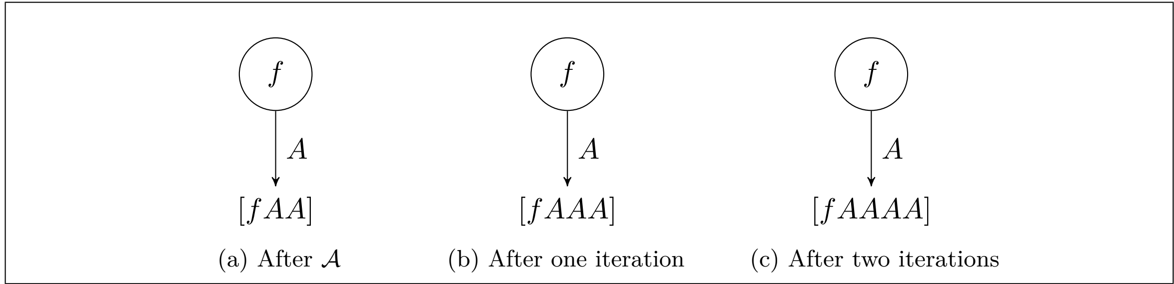


Figure 4.8.: Divergence in virtual nodes

does not jump to a local function, a new virtual node v', consisting of v plus the remaining arguments, is created and returned as the final node for this transition. As it is possible that a node reaches several virtual nodes with the same label, the final result also may return multiple virtual nodes.

If, during resolving a virtual node, the very same virtual node is passed again and there are still arguments to handle, the automata construction has to be aborted, because the current program flow diverges. This can be seen in an example in Fig. 4.8. Here the virtual node which is being resolved always adds one more instance of A to itself when looping through the automaton. This can be done infinitly many times and hence the resolving algorithm would never stop.

Naturally, this algorithm relies on the transition relation of the automaton, δ, to get the next state given a current state and a transition label. But δ only works on the exact pattern, i.e. if $(q, _, q') \in \delta$ and there exists no other transition from q, trying to find the next transition from q with the pattern A would fail. Furthermore, it needs to be specified what is the correct transition for label $A\ C$ if both $(q, _, p)$ and $(q, A\ _, p')$ are in δ. Therefore we define an enhanced transition relation $\hat{\delta}$ which choses the correct transition for each given pattern. The algorithm for $\hat{\delta}$ is given in Alg. 4.6.

The algorithm described in this section is given in Algorithms 4.7 and 4.8.

4.2.5. Correctness check

The automaton, which has been constructed by the presented rules and algorithms, does not need to be necessarily correct: The algorithms only checked for divergence in virtual nodes (Alg. 4.8) and correct arity of patterns (Alg. 4.3). Hence it is possible, that an invalid program passed the automata construction, but this automaton then is invalid too. For checking the validity of such an automaton, we define the following properties, which must hold:

```
Input: node q ∈ Q
Input: pattern p ∈ Σ ∪ {ε}
Output: node which is reached with p from q
  if p = ε then
    return δ(q, p)
  else
    labels := {l | ∃q′.(q, l, q′) ∈ δ ∧ p ⊑ l}
    -- sort labels lexicographically, where _ is seen as the largest letter
    slabels := sort labels
    -- now the first element of slabels is the most specific pattern which matches p
    return {q′ | (q, slabels[0], q′) ∈ δ}
  fi
```

Algorithm 4.6: Extended transition relation $\hat{\hat{\delta}}$

```
  for all v ∈ {v | v ∈ V ∧ ∃ l, q : (q, l, v) ∈ δ} do
    if name v ∈ dom(Φ) then -- local function
      parents := {q | ∃l. (q, l, v) ∈ δ}
      -- see algorithm 4.8 on the following page for the called function
      nodes := resolve v
      for all n ∈ nodes do
        for all p ∈ parents do
          for all l ∈ {l | (p, l, v) ∈ δ} do
            δ := δ \ {(p, l, v)}
            δ := δ ∪ {(p, l, n)}
          done
        done
      done
    fi
  done
```

Algorithm 4.7: Mapping virtual nodes

```
Input: virtual node v
Output: set of nodes v is replaced with
  nodes := {(v, [])}
  visited := v
  result := ∅
  alert := false

  while nodes is not empty do
    node, nargs := pop nodes
    if alert = true then
      fail "Cycle detected"
    else if node ∈ V then
      if n ∈ visited then
        alert := true
      else
        push node visited
      fi
    fi
    if node ∈ V ∧ name node ∉ Φ then
      result := result ∪ {[node + nargs]}
    else
      if n ∈ V then
        currargs := args node + nargs
        node := Φ(node)
      else
        currargs := args
      fi
      if currargs is not empty then
        trans := pop currargs
        next := δ̂(node, trans)
        nodes := nodes ∪ {(n, currargs) | n ∈ next}
      else -- nothing further to go to
        result := result ∪ {node}
      fi
    fi
  done
  return result
```

Algorithm 4.8: Resolving one virtual node

DFA-Check Our automaton has to be a deterministic finite automaton for the normal states in Q, and hence does not allow transistions under the same label from a certain state to a multiple other states:

$$\forall q \in Q \, \forall l \in \Sigma : \neg \exists x, y \in Q : (q, l, x) \in \delta \wedge (q, l, y) \in \delta \wedge x \neq y$$

No ε-transitions This kind of transition is a temporary one and must not exist in the final automaton:

$$\forall (x, l, y) \in \delta : l \neq \varepsilon$$

No edges between asterisk nodes Similar to the ε-transistions, these are only used temporarily and should have no connection to any other part of the automaton:

$$\forall (x, l, y) \in \delta : x \notin S \wedge y \notin S$$

If these three properties hold, the created automaton is deemed valid.

4.3. Constructor collection

The second part in this type inference algorithm is to collect the possible constructors of each variable and use them to construct the types of functions.

The constructor collection will again be presented in a syntax-directed style by giving rules for each syntactic construct. Such a rule will be of the form

$$\frac{\textit{premises}}{e \colon \bar{\tau} \rhd \tau \, \nabla \, U}$$

for each construct e.

This rule states that under certain premises, the expression e receives the type $\bar{\tau} \rhd \tau$ given that the set of constraints in U can be satisfied. The description of the meaning of the type $\bar{\tau} \rhd \tau$ has been given in Section 4.1 and applies here unaltered.

The constraint set U consists of subset relations of the form $\tau_1 \subseteq \tau_2$. Additionally, we write $\tau_1 = \tau_2$ as a short form for $\tau_1 \subseteq \tau_2, \tau_1 \supseteq \tau_2$, and omit the set braces when the set only contains one constructor.

Note that the rules often refer to the type by a symbolic τ and that the same τ in different positions (for example in the type of an expression and in the constraint set U) indeed refer to the very same type. These – though they look similar to type variables – are just labels for one specific type.

It is often common to have some sort of environment in the rule, which maps variable names to something specific in the rules (which would be types here). This showed to be clumsy for the comming rules, and therefore these rules assume that all variable names are unique. This assumption can easily be realised by the compiler, even for variables used but not declared in the current scope, and should hence be safe to make. And when variables are unique, it is also possible to assign a unique type label to each variable. From this follows, that there exists a function $V : \mathbb{V} \to \mathbb{T}$, that maps each variable name to the corresponding type.

Other type labels, which are neither retreived via V nor introduced with a premise, are always fresh. Explicitly declaring them as fresh is omitted for brevity.

4.3.1. Typing Rules

Similar to the other inference systems in this thesis, the difference between `end` and `skip` is small and only noticed in different behavior in match-statements. This behavior can be represented in types by using the unit constructor in the output type as has been described in Section 4.1.

$$\frac{}{\texttt{end}\colon \langle \tau_i | _ \rangle \rhd \tau_o \,\nabla\, \{\tau_o = ()\}} \quad \text{(CIP-End)}$$

$$\frac{}{\texttt{skip}\colon \langle \tau_o | _ \rangle \rhd \tau_o \,\nabla\, \emptyset} \quad \text{(CIP-Skip)}$$

Additionally, `skip` uses the same type for input and output type. This behavior allows the correct connection with the fall-through expression and will be illustrated with an example at the end of this section.

The next rule states the behavior for a constructor. It accounts for the implicit possibility of receiving a future (cf. Sec. 3.1.4). Also, similar to the other parts of the type inference system, it does not allow fall-through in a match and therefore uses () as the output type.

$$\frac{e_1 \colon \langle \tau_1 | _ \rangle \rhd \tau_1' \,\nabla\, U_1 \qquad \ldots \qquad e_n \colon \langle \tau_n | _ \rangle \rhd \tau_n' \,\nabla\, U_n}{C\ e_1 \ldots e_n \colon \langle \tau_C | _ \rangle \rhd \tau_C' \,\nabla\, \{\tau_C = \&(C\ \tau_1 \ldots \tau_n) \xrightarrow{\ -\ } \varepsilon, \tau_C' = ()\} \cup U_1 \cup \ldots \cup U_n} \quad \text{(CIP-Const)}$$

Referencing a variable and de-referencing a future work the probably expected way: The single variable reference only copies the type from the environment V and the future de-referencing adds a constraint requiring the type of the variable to be a future.

$$\frac{V(x) = \tau_x}{x \colon \langle \tau_x | _ \rangle \rhd \tau_o \,\nabla\, \{\tau_o = ()\}} \quad \text{(CIP-Var)}$$

$$\frac{V(x) = \tau_x}{!x\colon \langle \tau_i | _ \rangle \rhd \tau_o \,\nabla\, \{\tau_x \supseteq \&\tau_i, \tau_o = ()\}} \quad \text{(CIP-WAIT)}$$

Next we define the type of a lambda expression by getting each type of the expression inside the lambda and change it into a function constructor under the same label.

$$\frac{e\colon \langle \tau_{e_1} | P1, \ldots, \tau_{e_n} | Pn \rangle \rhd \tau'_e \,\nabla\, U \qquad V(x) = \tau_x}{\lambda\, x.\, e\colon \langle \tau_f | _ \rangle \rhd \tau'_e \,\nabla\, \{\tau_f = \{\tau_x \xrightarrow{P1} \tau_{e_1}, \ldots, \tau_x \xrightarrow{Pn} \tau_{e_n}\}\} \cup U} \quad \text{(CIP-LAMBDA)}$$

Similarly, a function application adds the constraint of a function with the appropriate types. The necessary pattern used in the function constructor is generated by the function $\texttt{pattern}\colon \mathbb{E} \to \mathbb{P}$. This function extracts the pattern syntacticly, this means that any variable implies the pattern _ and only statically defined constructors are taken into account. From this restriction follows a difference between the definitions $\texttt{let}\, f = g$ and $\texttt{let}\, f' = \lambda\, x.\, g\; x$. While f does not make any statement about any possible argument it is just an alias for g and all arguments are handled as if they were given to g directly. On the other hand f' makes all arguments be sent to g via the pattern _, which results from the use of the extra variable. So instead of being an alias, f' is a message relay explicitly receiving messages and sending them further to g – but with a possible loss of information.

$$\frac{e_1\colon \langle \tau_1 | _ \rangle \rhd \tau'_1 \,\nabla\, U_1 \qquad e_2\colon \langle \tau_2 | _ \rangle \rhd \tau'_2 \,\nabla\, U_2 \qquad P = \texttt{pattern}(e_2)}{e_1\; e_2\colon \langle \tau_i | _ \rangle \rhd \tau_o \,\nabla\, \{\tau_o = (), \tau_1 \supseteq \tau_2 \xrightarrow{P} \tau_i\} \cup U_1 \cup U_2} \quad \text{(CIP-APPLY)}$$

$$\frac{e\colon \langle \tau_e | _ \rangle \rhd \tau'_e \,\nabla\, U_e \qquad t\colon \bar{\tau}_i \rhd \tau_o \,\nabla\, U \qquad V(x) = \tau_x}{\texttt{letrec}\, x = e\, \texttt{in}\, t\colon \bar{\tau}_i \rhd \tau_o \,\nabla\, U_e \cup U \cup \{\tau_e = \tau_x\}} \quad \text{(CIP-LETREC)}$$

$$\frac{e\colon \langle \tau_e | _ \rangle \rhd \tau'_e \,\nabla\, U_e \qquad t\colon \bar{\tau}_i \rhd \tau_o \,\nabla\, U \qquad V(x) = \tau_x}{\texttt{let}\, x = e\, \texttt{in}\, t\colon \bar{\tau}_i \rhd \tau_o \,\nabla\, U_e \cup U \cup \{\tau_e = \tau_x\}} \quad \text{(CIP-LET)}$$

$$\frac{e\colon \langle \tau_e | _ \rangle \rhd \tau'_e \,\nabla\, U_e \qquad t\colon \bar{\tau}_i \rhd \tau_o \,\nabla\, U \qquad V(x) = \tau_x}{\texttt{bind}\, x = e\,;\; t\colon \bar{\tau}_i \rhd \tau_o \,\nabla\, U_e \cup U \cup \{\tau_x \supseteq \&\tau_e\}} \quad \text{(CIP-BIND)}$$

The three previous rules state the behavior of assignments, where CIP-LETREC and CIP-LET are identically. CIP-BIND differs in that it requires the bound variable to be a future. Additionally, the first two rules state equalities while binding only adds a subset constraint. This follows from the possibility to bind a future in different places where each place may

have a different type and all of those types must be collected.

$$\frac{t\colon \bar{\tau}_i \rhd \tau_o \nabla U \qquad f\colon \langle \tau_f|_\rangle \rhd \tau_f' \nabla U_f \qquad V(r) = \tau_r \qquad V(y) = \tau_y}{r, y \leftarrow f\,;\; t\colon \bar{\tau}_i \rhd \tau_o \nabla \{\tau_f \supseteq \tau_r \longrightarrow \tau_y\} \cup U \cup U_f} \quad \text{(CIP-Future)}$$

The rule for future sending is very similar to CIP-Apply. It does not even enforce the argument to be of a future type. This is caused by the way the types are constructed: The types themselves are sets of constructors defined by a set of subset relations. Hence an additional constraint $\tau_r \supseteq \&\tau_{rr}$ for some τ_{rr} does not result in an enhancement, because it is not possible for other places to reference the τ_{rr} and alter its constructor set.

This behavior conforms to the task of this part of the type inference: Only collect all possible constructors even if they are illegal. Rejecting invalid programs will then be done by the third part of the inference: the implication building.

The last rule to come covers the match-statement. It makes use of the two functions below:

- $\texttt{pattype}(y@p)$ – creates the constraints for each variable in $y@p$. So for instance

  $$\texttt{pattype}(y@(C\ x@(A\ B)\ z@(_)))$$

  will yield

  $$\{V(y) = \&(C\ V(x)\ V(z)) \to \varepsilon\} \cup \texttt{pattype}(x@(A\ B)) \cup \texttt{pattype}(z@_)$$

- $\uplus(\bar{\tau}_1, \bar{\tau}_2)$ – combines two constrained types by copying the types with unique patterns and unifying types with the same pattern. Therefore it returns the newly combined type and a set of constraints. Example:

  $$\biguplus(\langle \tau_{11}|_A, \tau_{12}|_B\rangle, \langle \tau_{21}|_A, \tau_{22}|_C\rangle) = \langle \tau_{11}|_A, \tau_{12}|_B, \tau_{22}|_C\rangle, \{\tau_{11} = \tau_{21}\}$$

The general idea is to create a constrained type $\hat{\bar{\tau}}_j$ for each branch with the pattern inferred by the automaton. These patterns are contained in $\mathcal{M}$. Using the automaton for this task allows to correctly handle refined matches without the overhead of tracking variables in these rules.

Such a constrained type is only newly created if and only if the type of the branch is constrained with the general pattern _. In all other cases, this type has already been created by nested matches inside the branch. As these hold the refined patterns, the current pattern is ignored.

Additionally the fall-through expression is added to the corresponding branches by adding the constraint $\tau'_j \subseteq \tau_t$ if output type τ'_j of the branch m_j is unequal to (). With this constraint, the type of the fall-through expression is stated to be equal to the output type of the last expression, which results in the change of the input type due to the definition of `skip` (see CIP-Skip above).

$$\frac{\begin{array}{c} U_{p_j} = \texttt{pattype}(y_j@p_j) \\ V(y_j) = \tau_j \qquad t_j : \bar{\tau}_j \rhd \tau'_j \,\nabla\, U_j \qquad \tilde{U}_j = \texttt{if } \tau'_j = () \texttt{ then } \emptyset \texttt{ else } \{\tau'_j \subseteq \tau_t\} \\ \hat{\bar{\tau}}_j = \texttt{case } \bar{\tau}_j \texttt{ of } \langle \tau | _ \rangle \texttt{ then } \langle \tau |_{\mathcal{M}(m_j)} \rangle \texttt{ otherwise } \bar{\tau}_j \\ \\ t\colon \langle \tau_t | _ \rangle \rhd \tau'_t \,\nabla\, U_t \qquad \bar{\tau}_i, \hat{U} = \biguplus_j \hat{\bar{\tau}}_j \qquad V(r) = \tau_r \qquad \forall j \in [1, n] \end{array}}{\begin{array}{c} \begin{pmatrix} ?r \,|_{m_1}\, y_1@p_1 \to t_1 \\ \vdots \\ |_{m_n}\, y_n@p_n \to t_n \end{pmatrix} ;\ t\colon \bar{\tau}_i \rhd \tau'_t \,\nabla\, \hat{U} \cup U_t \cup \\ \\ \bigcup_{j \in [1,n]} (\{\tau_r \supseteq \&\tau_j \Longrightarrow \varepsilon\} \cup U_j \cup U_{p_j} \cup \tilde{U}_j) \end{array}} \quad \text{(CIP-Match)}$$

The type of the overall match is obtained by combining the constrained types of each branch into one constrained type (via the $\biguplus$ function). As this unifies these branches with the same pattern, it correctly differentiates matches on the last message, where each branch has an explicit pattern, from other matches, where no real branching happens and all branches are given the pattern _. Hence all these branches and therefore their types are unified.

4.3.2. Combining the Types / Modularity

The inference of the previous section is, as described in the introduction to this chapter, ran on each global function in itself. Thereafter the constraint sets of the different functions are combined into one large set. Because of the use of unique variable names and also unique type names, references to external functions in a definition can be resolved automatically in this large set, as the type names match.

This can be done multiple times to create even larger sets – so for example first one module, then different modules into one package and finally different packages. By saving the current constraint set the combination process can also be executed at a later point in time as no information is lost.

Depending on the definition of *modularity*, this behavior may not qualify as modular, because the exact type of a function depends on all uses. For this to change, the introduction

of type variables is necessary. This allows to only mention these constructors in the type of a function, which are explicitly given (most often as part of a match). The type would then get extended by a type variable and all constructors not explicitly mentioned would map onto this type variable and have the same flow through the function.

This extension is not presented in this thesis and should be part of future work.

4.4. Implication analysis

After the types of each variable have been determined in the previous section, this section will show a method to generate a set of implications between different occurrences of constructors. This allows to model the flow of type information through the program. Hence certain constructors that cannot be reached are illegal and thus it is possible to reject invalid programs. This is also possible by solving the created system of implications: If no solution can be found, the program is invalid. Because the flow of constructors is known, it is then possible to give detailed output where the source of error lies and which path through the program leads there.

The general idea of this approach is to attach a unique label to each occurrence of a constructor. Then, using a syntax-oriented inference system, implications can be created between these occurrences that imply another. Due to the branching, these implications do not only exist between different instantiations of the same constructor but also between different constructors.

The whole process is best illustrated by an example: Assume the code given in Fig. 4.9. By using the information gathered from collecting the constructors, all variables are replaced by their types, i.e. the set of constructors they represent. This might lead to a program as given in Fig. 4.10. Between these constructors, we create the implications yielding the following set of implications:

$$\{A_2 \Rightarrow A_1, A_1 \Rightarrow A_7, A_3 \Rightarrow A_4, B_1 \Rightarrow B_2, A_4 \Rightarrow A_5,$$
$$A_5 \Rightarrow A_6, B_2 \Rightarrow B_3, B_3 \Rightarrow C_1, A_7 \Rightarrow A_3\}$$

This result is, of course, a simplified version of what the presented rules would infer. For instance, the function symbols in the examples were not touched, while they would have been substituted by their types too. Also the flow back from g to f (e.g. rules from A_6 and C_1 to specify the return value of f) is omitted.

For this inference algorithm we again assume, that each variable is unique. Additionally it is assumed, that each expression is annotated with its type. This can for example be done while doing the constructor set inference. And a third assumption is, that variables which

```
        let f = let y = A in
                let g = \x -> x | A -> x
                                | B -> C
                in g A
```

Figure 4.9.: Example program

```
        let f = let A1 = A2 in
                let g = \{A3,B1} -> {A4,B2} | A5 -> A6
                                            | B3 -> C1
                in g A7
```

Figure 4.10.: Example program after substituting types

are matched against are not used inside the branches. Instead the outermost variable of the varpattern of this branch is to be used:

```
x | y@A -> -- only use y here
  | y@(B C) -> -- only use y here
  | z@_ -> -- only use z here
```

The reason lies in the different set of constructors bound to these variables: The variable which is matched, does necessarily contain all possible choices of the branches. The types of the outermost variables in the varpattern however only contain the constructors for their respective branch. As both are equal in behavior inside the branch and the types of the varpattern variables are just subsets of the type of the matched variable, this is a safe program transformation. Even more, with the previously given assumption of unique variable names, the replacement can happen without accidently replacing other variables with the same name.

Similar to the other sections of this chapter, we will first introduce some notation that will be used throughout this section. Thereafter we will give the syntax-directed rules.

4.4.1. Notation

In this section, we define, in addition to the other sets $\mathbb{V}$, $\mathbb{T}$ etc., a set $\mathbb{C}_A$ to be the set of all annotated constructors, which are built by the following grammar, where l is any unique label.

$$ca ::= cons_l\ ca_1 \ldots ca_n \quad (n \geq 0)$$

The powerset of $\mathbb{C}_A$ yields the set of annotated types $\mathbb{T}_A = 2^{\mathbb{C}_A}$, as types are sets of constructors. With defining the absence of annotations as annotations with the empty string,

it follows $\mathbb{T} \subset \mathbb{T}_A$.

Furthermore we define a substitution $\sigma\colon \mathbb{T}_\mathbb{A} \to \mathbb{T}_A$, that takes a type (be it annotated or not) and returns a set of annotated types, i.e. it annotates each constructor in the passed type. If the type is already annotated, the annotation is replaced by the new one. By definition, the annotations are unique and hence multiple applications with the same argument return different results. For ease of notation, we sometimes use the notion of adding subscripts to σ, for instance σ_1. This shall partly revoke the previous statement, as multiple applications of the same argument to a σ with subscript returns the same result:

$$\sigma(A) \neq \sigma(A)$$

$$\sigma_1(A) = \sigma_1(A) \neq \sigma_2(A) \neq \sigma(A)$$

Each of the upcoming rules for an expression e will then be of the form:

$$\frac{\textit{premises}}{\Gamma, G \vdash e : I \triangleright G', \mathfrak{T}}$$

with the following denotation of the variables used:

- $\Gamma\colon \mathbb{V} \to \mathbb{T}_A$: an environment mapping the variable to its annotated type
- G, G': a logical formula (*guard*) consisting of branch labels $\in \mathbb{M}$ and False. The branch labels in this context are normal atoms of the formula without any special functionality. They are used in the implications for stating that the pattern of the branch has been matched and this implication therefore uses this branch. While G is the formula before evaluating the rule, G' is valid after the rule.
- I: a set of implications. The atoms of the implications are annotated constructors (without arguments) and branch labels, while the implications in the set itself are considered to be connected by conjunctions.
- $\mathfrak{T} \in \mathbb{T}_A$: the annotated type of the current expression. Because σ returns unique constructors, they need to be explicitly passed around. Otherwise it would not be possible to re-create them.

The rules do not create the set of implications explicitly, because if a constructor contains &, the direction of the implication has to be reversed. This happens because the values bound to futures "flows" all the way back to where it was sent. Therefore the creation of implications is done by the function $\texttt{imp}(G, l, r)$. This function generates a set of

implications from each constructor in l to each constructor in r with the guard-set G. This is done recursively for each argument. If a reference is encountered, the direction is reversed once. For example $\texttt{imp}(G, \{C_1\ (A_1\ \&C_3)\}, \{C_2\ (A_2\ \&C_4)\})$ would generate $\{G \wedge C_1 \rightarrow C_2, G \wedge A_1 \rightarrow A_2, G \wedge C_4 \rightarrow C_3\}$.

The inference is initialized with $G = \mathsf{True}$ and Γ mapping each global function to its annotated type.

4.4.2. Implication Rules

Again, `skip` and `end` only differ in that the latter prevents fall-through in match-statements. This is achieved by having False as the resulting guard-formula which results in implications of the form $\mathsf{False} \wedge \ldots \Rightarrow \ldots$. Such an implication is a neutral element in the set of implications, while it itself stops the flow of constructors to the constructors of the fall-through statement which would be found in the consequent.

$$\frac{}{\Gamma, G \vdash \texttt{skip}\colon \emptyset \rhd G, \emptyset} \quad \text{(IMP-Skip)}$$

$$\frac{}{\Gamma, G \vdash \texttt{end}\colon \emptyset \rhd \mathsf{False}, \emptyset} \quad \text{(IMP-End)}$$

The rule for constructors again only cumulates the arguments of the constructor and creates the function representation of the constructor. Note that this rule explicitly states an implication. This is because the constructor is freshly introduced and therefore, besides the guard, there is no annotated constructor that could be part of the antecedent.

$$\frac{\Gamma, G \vdash e_i\colon I_i \rhd G'_i, \mathfrak{T}_i \qquad i \in [1, n]}{\begin{array}{c}\Gamma, G \vdash C\ e_1 \ldots e_n\colon \{G \Rightarrow \sigma_1(C)\} \cup I_1 \cup \ldots \cup I_n \rhd \\ \mathsf{False}, \{\&(\sigma_1(C)\ \mathfrak{T}_1 \ldots \mathfrak{T}_n) \longrightarrow \emptyset\}\end{array}} \quad \text{(IMP-Const)}$$

Variables create implications between the current annotated type and a newly annotated one.

$$\frac{}{\Gamma, G \vdash x\colon \{\texttt{imp}(G, \Gamma(x), \sigma_1(\Gamma(x)))\} \rhd \mathsf{False}, \sigma_1(\Gamma(x))} \quad \text{(IMP-Var)}$$

Dereferencing futures behaves similarly, but instead of the whole type it uses the part of

the type inside the future.

$$\frac{}{\Gamma, G \vdash !x\colon \{\mathtt{imp}(G, c, \sigma_1(c)) \mid \&c \in \Gamma(x)\} \rhd \mathsf{False}, \{\sigma_1(c) \mid \&c \in \Gamma(x)\}} \quad \text{(IMP-Wait)}$$

It has been an assumption noted in the introduction to this section, that each expression has explicitly stated its type. This can be seen in the rule IMP-Lambda, where the lambda expression is assigned its type τ_λ. For reasons of brevity, this type will only be stated when it is referred to in the rule itself. It must also be noted, that the usage of τ_λ is not completely accurate, as the full type would be something like $\langle \tau_\lambda | _ \rangle \rhd \tau_o$ as stated in the previous section. But the output type is of no relevance anymore, as it was only needed to generate a correct equation system. Hence it can be dropped here. Further we will only use explicit types in cases, where there is no constraint added. Therefore it also possible to strip the shell of the constrained type and only use the τ_λ inside.

Now this type is used to generate an annotated version by annotating argument and result type. Additionally implications are generated from the annotated type of the expression inside the lambda to the different result types.

$$\frac{\tau_\lambda = \{X \xrightarrow{P_1} E_1, \ldots, X \xrightarrow{P_n} E_n\} \qquad \Gamma[x \mapsto \sigma_1(X)], G \vdash e\colon I \rhd G', \mathfrak{T}}{\Gamma, G \vdash (\lambda\, x.\, e) : \tau_\lambda\colon \bigcup_i \{\mathtt{imp}(G, \mathfrak{T}, \sigma_2(E_i))\} \cup I \rhd G', \bigcup_i \{\sigma_1(X) \xrightarrow{P_i} \sigma_2(E_i)\}} \quad \text{(IMP-Lambda)}$$

IMP-Apply then goes the opposite way and adds implications from the annotated type of the argument to the argument of the function. Further it defines the result type of the whole application to be the union of all possible result types. There is no need to differentiate by pattern as the function itself generates the correct implications based on the argument. Thus through a chain of implications, the argument constructors influence the constructors in the result.

$$\frac{\begin{array}{c}\Gamma, G \vdash e_1\colon I_1 \rhd G_1', \mathfrak{T}_1 \qquad \mathfrak{T}_1 = \{X \xrightarrow{P_1} E_1, \ldots\} \\ \Gamma, G \vdash e_2\colon I_2 \rhd G_2', \mathfrak{T}_2\end{array}}{\Gamma, G \vdash e_1\ e_2\colon I_1 \cup I_2 \cup \{\mathtt{imp}(G, \mathfrak{T}_2, X)\} \rhd \mathsf{False}, \bigcup_i E_i} \quad \text{(IMP-Apply)}$$

$$\frac{\Gamma[x \mapsto \sigma_1(\tau_x)], G \vdash t\colon I \rhd G', \mathfrak{T} \qquad \Gamma, G \vdash e\colon I_e \rhd G_e', \mathfrak{T}_e}{\Gamma, G \vdash \mathtt{let}\, x : \tau_x = e\, \mathtt{in}\, t\colon \{\mathtt{imp}(G, \mathfrak{T}_e, \sigma_1(\tau_x))\} \cup I \cup I_e \rhd G', \mathfrak{T}} \quad \text{(IMP-Let)}$$

$$\frac{\Gamma_x = \Gamma[x \mapsto \sigma_1(\tau_x)] \qquad \Gamma_x, G \vdash t \colon I \rhd G', \mathfrak{T} \qquad \Gamma_x, G \vdash e \colon I_e \rhd G'_e, \mathfrak{T}_e}{\Gamma, G \vdash \mathtt{letrec}\, x : \tau_x = e\, \mathtt{in}\, t \colon \{\mathtt{imp}(G, \mathfrak{T}_e, \sigma_1(\tau_x))\} \cup I \cup I_e \rhd G', \mathfrak{T}} \quad \text{(IMP-LETREC)}$$

The two previous rules for assignment added implications from the assigned values to the variable. The same is done for the binding operation in the next rule. But the difference here lies in a subtle alternation: While IMP-LET and IMP-LETREC created new assignments for the constructor sets, IMP-BIND uses the same type that is already existing in the environment. Moreover, by the way `imp` is invoked, the implication runs from the introduction of the variable (probably in the lambda expression) to the bound expression. But as they both contain & (it is added explicitly for the bound expression in $\mathfrak{T}'_e$), these implications are reversed. And hence the constructors of the bound expression indeed point to the future in the lambda expression. And this indeed is the desired behavior, as the flow of information for futures is backwards.

$$\frac{\Gamma \setminus x, G \vdash t \colon I \rhd G', \mathfrak{T} \qquad \Gamma, G \vdash e \colon I_e \rhd G'_e, \mathfrak{T}_e \qquad \mathfrak{T}'_e = \{\&c \mid c \in \mathfrak{T}_e\}}{\Gamma, G \vdash \mathtt{bind}\, x = e\,;\ t \colon \{\mathtt{imp}(G, \Gamma(x), \mathfrak{T}'_e)\} \cup I \cup I_e \rhd G', \mathfrak{T}} \quad \text{(IMP-BIND)}$$

Similar considerations have been made for IMP-FUTURE: Here $\sigma_1(X)$ seems to link to X, while both contain & and therefore the implication is reversed.

$$\frac{\begin{array}{c}\Gamma, G \vdash e \colon I_e \rhd G'_e, \mathfrak{T}_e \qquad \mathfrak{T}_e = \{X \longrightarrow Y\} \\ \Gamma[r \mapsto \sigma_1(X), y \mapsto \sigma_2(Y)], G \vdash t \colon I \rhd G', \mathfrak{T}\end{array}}{\Gamma, G \vdash r, y \leftarrow e\,;\ t \colon \{\mathtt{imp}(G, \sigma_1(X), X), \mathtt{imp}(G, Y, \sigma_2(Y))\} \cup I \cup I_e \rhd G', \mathfrak{T}} \quad \text{(IMP-FUTURE)}$$

Finally, we have the rule that handles the match-statement. This rule uses a special function $\mathtt{patcons}(y@p, r, c)$. For each variable in the varpattern $y@p$ it generates the constructor set of the matched sub-pattern and returns a mapping Γ_i from these variables to the corresponding annotated constructors. It is therefore quite similar to the `createprocs` function used during the operational semantics (Fig. 3.2). It further returns a set of implications I_{y_i} from the c to these constructors.

The guard formula inside the branch does not hold the set of constructors of r that were needed to reach this branch. Instead it simply adds m_i, i.e. the label of the branch. And in I_{p_i} the constructors imply that label. Here it must be taken care, that only these constructors are taken into account, which were not matched by preceding matches. Otherwise the same constructors that, for example matched $A\ B$ will also match A _ and hence create a disambiguity.

Finally, the resulting guard formulas of each branch are put into a disjunction to create the guard for the fall-through case. Thereby, again the different fall-through behavior of the branches is correctly represented.

$$\frac{\begin{array}{c}\Gamma_i, I_{y_i} = \texttt{patcons}(y_i@p_i, r, G \wedge m_i) \qquad \Gamma \circ \Gamma_i, G \wedge m_i \vdash t_i \colon I_i \rhd G'_i, \mathfrak{T}_i \\ I_{p_i} = \{\texttt{imp}(G, c_i, m_i) \mid \&c_i \in \Gamma(r), c_i \sqsubseteq p_i, \forall \hat{p} \in \{p_1, \ldots, p_{i-1}\} \colon c_i \not\sqsubseteq \hat{p}\} \qquad \forall i \in [1, n] \\ \\ \Gamma, G \wedge (G'_1 \vee \ldots \vee G'_i) \vdash t \colon I_t \rhd G'_t, \mathfrak{T}_t\end{array}}{\Gamma, G \vdash \begin{pmatrix} ?r \,|_{m_1}\, y_1@p_1 \rightarrow t_1 \\ \vdots \\ |_{m_n}\, y_n@p_n \rightarrow t_n \end{pmatrix};\ t \colon \bigcup_i (I_i \cup I_{p_i} \cup I_{y_i}) \cup I_t \rhd G'_t, \bigcup_i \mathfrak{T}_i \cup \mathfrak{T}_t} \text{(IMP-Match)}$$

5. Conclusion

During this work we showed a novel approach for a type system and its inference. It was aimed on tackle certain short-comings in the Hindley-Milner system. These were:

untypable correct expression In the presented systems this should not happen, though those yielding infinite types are still not possible.

missing modularity Depending on the definition on *modularity*, this might be resolved.

unrelated error messages The core of the type systems is the modeling of type flow. Hence it is possible to exactly give the path that lead to a certain type error.

branch-aware functions This part is another core of the type system and hence is successfully implemented.

The points, where we cannot explicitly state, whether the goal is fulfilled are in need of type variables. These are currently not part of the system and therefore restrict the application, as all types have to be collected instead of using an abstract variable to hold all uncollected constructors.

A further drawback of the current system, which needs further research, comes from disallowing infinite types. Here it needs investigation if the system can be extended to also support them, or if they are not needed and can for example be expressed using processes, as processes can be infinite without any problems.

And while we assume, that this system is both sound and complete, these properties need a formal proof.

Besides the type system we showed a process-based concurrent language. By using the notion of processes, this language allows constructs which are not possible in conventional functional languages.

A. Grips language

A.1. Syntax

$$
\begin{array}{rll}
expr ::= & \texttt{skip} & \text{(Empty expression)} \\
| & \texttt{end} & \text{(Empty expression \#2)} \\
| & var & \text{(Variable call)} \\
| & C\ expr_1 \ldots expr_n & \text{(Constructor, } n \geq 0\text{)} \\
| & \lambda\, var.\, expr & \text{(Abstraction)} \\
| & expr_1\ expr_2 & \text{(Application)} \\
| & \texttt{let}\, var = expr_1\, \texttt{in}\, expr_2 & \text{(Let-Binding)} \\
| & \texttt{letrec}\, var = expr_1\, \texttt{in}\, expr_2 & \text{(Recursive Let-Binding)} \\
| & \texttt{bind}\, var = expr_1\ ;\ expr_1 & \text{(Future binding)} \\
| & var_1, var_2 \leftarrow expr_1\ ;\ expr_2 & \text{(Future sending)} \\
| & !var & \text{(Wait)} \\
| & \begin{array}{l} ?var\ |_{m_1}\ var_1@varpat_1 \rightarrow expr_1 \\ \quad \vdots \qquad\qquad ;\ expr \\ \quad |_{m_n}\ var_n@varpat_n \rightarrow expr_n \end{array} & \text{(Pattern Matching)}
\end{array}
$$

$$
\begin{array}{rll}
pat ::= & _ & \text{(Matches everything)} \\
| & C\ pat_1 \ldots pat_n & \text{(Matches constructor } C \text{ and tests all arguments)}
\end{array}
$$

$$
\begin{array}{rl}
varpat ::= & var\,@\,_ \\
| & var\,@\,C\ varpat_1 \ldots varpat_n
\end{array}
$$

A.2. Syntactic sugar

Two general rules apply:

- variables which are not needed can be omitted and fresh anonymous variables will be inserted in place
- where the syntax demands `in` *expr* or ; *expr*, the *expr* can be omitted and is taken as being ε

$$
\begin{array}{rcl}
\begin{array}{r} s_1 \\ s_2 \end{array} & \equiv & s_1\,;\; s_2 \\
f x_1 \ldots x_n = t & \equiv & \texttt{letrec}\, f = \lambda\, x_1.\;\; \ldots \lambda\, x_n.\, t\, \texttt{in} \\
\begin{array}{r} f\ C = t_1 \\ f\ D = t_2 \end{array} & \equiv & \begin{array}{r} \texttt{letrec}\, f = \lambda\, y.\, y \mid C \rightarrow t_1 \\ \mid D \rightarrow t_2 \end{array} \texttt{in} \\
e \mid r, x' \rightarrow t & \equiv & r, x' \leftarrow e\,;\; t \\
e \mid y@pat, e' \rightarrow t & \equiv & (r, e' \leftarrow e\,; ?r \mid y@pat \rightarrow t) \text{ for some fresh } r \\
\texttt{return}\, x & \equiv & \lambda\, res.\, \texttt{bind}\, res = x \\
\backslash x \rightarrow t & \equiv & \lambda\, x.\, t
\end{array}
$$

Bibliography

[Arm03] Armstrong, Joe: *Making reliable distributed systems in the presence of software errors*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, December 2003.

[Bar90] Barendregt, Hendriek Pieter: *Functional programming and lambda calculus*. In Leeuwen, Jan van (editor): *Handbook of theoretical computer science*, volume B, pages 321–363. MIT Press, Cambridge, MA, USA, 1990.

[Car97] Cardelli, Luca: *Type Systems*. In Tucker, Allen B. (editor): *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, 1997.

[Chi01] Chitil, Olaf: *Compositional explanation of types and algorithmic debugging of type errors*. SIGPLAN Not., 36:193–204, October 2001.

[CW85] Cardelli, Luca and Peter Wegner: *On Understanding Types, Data Abstraction, and Polymorphism*. ACM Computing Surveys, 17(4):471–522, 1985.

[DM82] Damas, Luis and Robin Milner: *Principal type-schemes for functional programs*. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.

[GJSB05] Gosling, James, Billy Joy, Guy Steele and Gilad Bracha: *The Java Language Specification*. Addison-Wesley Professional, 2005.

[Hal85] Halstead, Jr., Robert H.: *MULTILISP: a language for concurrent symbolic computation*. ACM Trans. Program. Lang. Syst., 7:501–538, October 1985.

[Hin69] Hindley, R.: *The Principal Type-Scheme of an Object in Combinatory Logic*. Transactions of the American Mathematical Society, 146:29–60, 1969.

[HMU00] Hopcroft, John E., Rajeev Motwani and Jeffrey D. Ullman: *Introduction to Automata Theory, Languages and Computability*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000.

[Hoa78] HOARE, C. A. R.: *Communicating sequential processes.* Commun. ACM, 21:666–677, August 1978.

[Kah87] KAHN, GILLES: *Natural semantics.* In BRANDENBURG, FRANZ, GUY VIDAL-NAQUET and MARTIN WIRSING (editors): *STACS 87*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer Berlin / Heidelberg, 1987. 10.1007/BFb0039592.

[KR88] KERNIGHAN, BRIAN W. and DENNIS M. RITCHIE: *The C Programming Language.* Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[Mil78] MILNER, ROBIN: *A theory of type polymorphism in programming.* Journal of Computer and System Sciences, 17(3):348 – 375, 1978.

[MTHM97] MILNER, ROBIN, MADS TOFTE, ROBERT HARPER and DAVID MACQUEEN: *The Definition of Standard ML (Revised).* The MIT Press, 1997.

[MW97] MARLOW, SIMON and PHILIP WADLER: *A practical subtyping system for Erlang.* In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 136–149, New York, NY, USA, 1997. ACM.

[Myc84] MYCROFT, ALAN: *Polymorphic type schemes and recursive definitions.* In PAUL, M. and B. ROBINET (editors): *International Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228. Springer Berlin / Heidelberg, 1984.

[Pie02] PIERCE, BENJAMIN C.: *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002.

[PJ03] PEYTON JONES, SIMON (editor): *Haskell 98 Language and Libraries – The Revised Report.* Cambridge University Press, 2003.

[Plo81] PLOTKIN, GORDON D.: *A Structural Approach to Operational Semantics.* Technical Report DAIMI FN-19, University of Aarhus, 1981.

[pyt11] *The Python Language Reference.* `http://docs.python.org/reference/`, March 2011.

[Rey98] REYNOLDS, JOHN C.: *Theories of Programming Languages.* Cambridge University Press, New York, NY, USA, 1st edition, 1998.

[Ros98] ROSCOE, A. W.: *The Theory and Practice of Concurrency.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998.

[SP07] Simonet, Vincent and François Pottier: *A constraint-based approach to guarded algebraic data types.* ACM Transactions on Programming Languages and Systems, 29, January 2007.

[SS71] Scott, Dana and Christopher Strachey: *Toward A Mathematical Semantics for Computer Languages.* In Fox, Jerome (editor): *Proceedings of the Symposium on Computers and Automata*, volume XXI, pages 19–46, Brooklyn, N.Y., April 1971. Polytechnic Press.

[SS86] Sterling, Leon and Ehud Shapiro: *The Art of Prolog: Advanced Programming Techniques.* The MIT Press, 1986.

[Tof88] Tofte, Mads: *Operational Semantics and Polymorphic Type Inference.* PhD thesis, University of Edinburgh, 1988.

[VR03] Van Rossum, Guido: *The Python Language Reference Manual.* Network Theory Ltd., September 2003.

[WB89] Wadler, P. and S. Blott: *How to make ad-hoc polymorphism less ad hoc.* In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.

[Wel02] Wells, J. B.: *The Essence of Principal Typings.* In Widmayer, Peter, Stephan Eidenbenz, Francisco Triguero, Rafael Morales, Ricardo Conejo and Matthew Hennessy (editors): *Automata, Languages and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 774–774. Springer Berlin / Heidelberg, 2002.

[XP99] Xi, Hongwei and Frank Pfenning: *Dependent types in practical programming.* In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 214–227, New York, NY, USA, 1999. ACM.