UNIVERSITY OF GOTHENBURG

$$\begin{tikzcd}
 & F_\omega \arrow[rr] & & FP_\omega \\
F \arrow[rr] \arrow[ur] & & FP \arrow[ur] & \\
 & \lambda_\omega \arrow[rr] \arrow[uu] & & \lambda P_\omega \arrow[uu] \arrow[ulll, red] \\
\lambda_\rightarrow \arrow[rr] \arrow[ur] \arrow[uu] & & \lambda P \arrow[ur] \arrow[uu] &
\end{tikzcd}$$

Compiling Agda to System Fω in Theory

Bachelor of Science Thesis in Software Engineering and Management

GREGOR ULM

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, June 2015

Compiling Agda to System Fω in Theory
Gregor Ulm

Examiner: Morgan Ericsson

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover: The cover image shows Barendregt's λ-cube, with an added red relation arrow. The λ-cube is a conceptual framework that illustrates the relations between various typed λ-calculi. The origin of the red relation arrow locates Agda within the λ-cube, while its tip points to System Fω.

Department of Computer Science and Engineering
Göteborg, Sweden, June 2015

Compiling Agda to System F_ω in Theory

Gregor Ulm
University of Gothenburg
gregor.ulm@gmail.com

Abstract

We develop a theoretical foundation for compiling the programming language Agda to System F_ω, which is a stepping stone towards a compiler from Agda to Haskell. The practical relevance for software engineering and the problem of providing correctness guarantees for programs is highlighted. After describing relevant λ-calculi, we specify the semantics for compiling Agda to System F_ω. Finally, we illustrate those compilation rules by manually translating several Agda code examples to System F_ω.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Languages, Theory

Keywords Languages, Lambda calculus and related systems, Types, Compilers

1. Introduction

Agda is a dependently-typed functional programming language as well as a proof assistant. Currently, Agda programs can be interpreted, but the various compilation backends are not fully satisfactory yet. There is a compiler backend that targets Haskell, MAlonzo, which uses the untyped λ-calculus as an intermediate language. Therefore, it has to rely on type coercions for the translation to Haskell, which is based on a type system that is closely related to System F_ω. This means that there is little opportunity for any optimisations to be performed by the Haskell compiler GHC.

Consequently, there is room to improve the *status quo* by creating a new compiler backend that performs type-directed translation and resorts to type coercion only when strictly necessary. Presenting a set of rules for the extraction from Agda to System F_ω, which is intended as the theoretical foundation of a new compiler backend for Agda, is the aim and culmination of this paper. Some preliminaries are necessary before we reach that point. The translation to System F_ω, however, is only a stepping stone towards translating Agda to Haskell, as the necessary sequence of translations goes from Agda to System F_ω, and from System F_ω to Haskell.

The context of this paper is a Bachelor's thesis for a degree in Software Engineering and Management. This warrants taking great care to highlight, in Section 2, the significance of our work for software engineering. We are stressing the relation between the necessity of tests for programs written in a particular programming language, and the strength of the type system it uses. Our presentation illustrates that there is an inverse correlation between the expressiveness of a type system and the need to write test cases, i.e. the more expressive the type system of a given programming language, the fewer cases have to be manually specified, due to the fact that more expressive type systems provide correctness guarantees for free — from the view of the working programmer at least. We discuss dynamic typing, static typing, and dependent typing, as well as some of their respective benefits for the working programmer.

In Section 3 we embark on a brisk tour of several important λ-calculi, which is necessary for providing the background for the extraction rules from Agda to System F_ω. We will provide an illustration of the untyped λ-calculus, the simply typed λ-calculus, the polymorphic λ-calculus and the simply typed λ-calculus with type operators. While being far from exhaustive, our presentation nonetheless aims to provide the reader with a sufficient description of the various λ-calculi, important operations and rules, and the motivations that led to their development by highlighting the respective problems they were designed to solve.

Section 4 is devoted to the translation of programming languages. After discussing some relevant theoretical background, we continue with a specification of the higher-order polymorphic λ-calculus, i.e. System F_ω, and Agda, before we, at long last, describe the translation from Agda to System F_ω, which forms the main contribution of this paper. Concretely, we do the following:

- locate Agda within the λ-cube, which is a conceptual framework for classifying various λ-calculi
- present a slightly simplified specification of Agda as a suitable core for a new compiler
- extend System F_ω in order to make it an appropriate target for Agda
- specify the translation from Agda to System F_ω by giving the semantics of the compilation rules
- illustrate the translation by providing several examples that show the extraction from Agda to System F_ω

Lastly, in Section 5 we provide an outlook to potential future work that may build upon the results presented in this paper. We highlight the relevance of a compiler from Agda to System F_ω, which is a step towards a compiler that targets a practical general-purpose programming language like Haskell. In this context, we also discuss why it may be desirable, for practical reasons, to compile an Agda program into a Haskell program.

2. Relevance for the software engineering discipline

2.1 The cost of fixing software defects

Practitioners in the software engineering domain may view programming language theory in general, and type theory in particular, as one of the least applicable subfields within computer science. On the other hand, there is a strong consensus within software engineering that it is of tantamount importance to create software that contains as few critical defects as possible. From a business point of view this is understandable, considering that the longer software defects remain in the code base, the more costly they are to fix. For instance, fixing a software defect before the source code has been committed to a code repository may be trivial, while fixing a software defect in a production system is anything but.

Commonly purported figures are increases in cost by powers of ten. Illustrated by reference to the sequential waterfall software development methodology, Patton [36] claims that there is a tenfold increase of costs at each phase of a software development project. A software defect that would have cost $1 to fix in the specification phase will eventually, after not having been discovered in the design phase, implementation phase, and testing phase, cost $10,000 to fix in the release phase. Empirical numbers are not quite as tidy, however. For instance, in a meta-study on the costs of fixing software defects, Shull et al. [41] point out that a roughly 100-fold increase between the phases "code to test" and "test to field", to use their terminology, could be verified for critical defects. On the other hand, for non-critical defects the ratio may be as little as 2 to 1, instead of an order of a magnitude. This is still a significant difference, but it is not quite as dramatic.

2.2 Limitations of unit testing

Unit testing is a popular approach to ensure the absence of defects, with some degree of uncertainty [40]. As the name indicates, it is comprised of tests that aim to verify *units* of source code, which are tested with a finite number of non-random inputs. Within software engineering, the reliance on unit testing has become so predominant that an entire school of thought, test-driven development (TDD), has sprung up that aims to use tests as a guide for software development [6].

However, unit testing is rather unsatisfying from a methodological perspective, as it is akin to attempting a mathematical proof by means of incomplete enumeration or by example. Unit tests do not verify that the tested units of source code are correct. Instead, they only provide some degree of assurance that the tested units of source code are correct for the provided inputs. However, in programming languages with mutable state, unit testing might not be much of an assurance at all, given that identical function calls may lead to different outputs.

A more promising approach than unit testing is property-based testing, for instance by using QuickCheck [12]. While QuickCheck was originally developed for testing Haskell programs, a commercial version is available that allows testing of programs in Erlang and C. The idea behind QuickCheck is to define mathematical properties that are supposed to hold for all inputs. Those properties are subsequently tested by randomly generating test cases of increasing complexity. Due to this approach, QuickCheck is able to uncover bugs that would be next to impossible to track down with unit tests.

2.3 Testing and type systems

When viewed abstractly, it seems that testing is little more than an attempt to overcome limitations of the target programming language. One might expect that the testing efforts that are necessary in a particular programming language are inversely proportional to the expressiveness of its type system. Test cases for basic properties that could very well be verified by the type checker in a programming language with a more expressive type system will have to be tested manually in a programming language with a less expressive type system. Several source code examples that illustrate this point are given further below. This leads to the observation that programming languages with more expressive type systems are safer because they protect their own abstractions, as Pierce [37, p. 6] so succinctly expresses it. Conversely, unsafe programming languages do not protect their own abstractions.

To give some examples of those weaknesses: In the programming language C [23], memory locations can be accessed freely, which has the side effect of sustaining a large part of the computer security industry. Dynamically typed languages like Python [44] or Ruby [27] do not perform any checks at all prior to execution.[1] Even in a programming language with a type system that is as advanced as the one used by Haskell, tests may be necessary to verify that certain expected properties do indeed hold. A popular example is sorting. For instance, QuickCheck properties could be used to confirm that the output of a sorting function is indeed sorted.

Perhaps surprisingly for software engineers who are unacquainted with the *avant-garde* of programming language technology, there are advanced programming languages that make it possible for the programmer to encode specifications as types. For instance, in the dependently typed programming language Agda [33] it is possible to encode the desired property that the output of a sorting function is indeed sorted, as part of the type signature. This means that the type checker will attempt to verify this property, and reject programs for which it does not hold.

2.4 Some practical examples

As was stated above, type systems help to ensure program correctness. The more expressive the type system of a given programming language, the more guarantees can be made about a program written in it. In practical terms this means that there is an inverse correlation between the effort that is necessary for testing and the expressiveness of the type system that is used. To illustrate this inverse correlation, we describe a relatively simple sorting algorithm in pseudocode below, and subsequently translate it into Python, Java, and Haskell, which are programming languages with increasingly more expressive type systems.

Let us take the pseudo-code definition of insertion sort, following the presentation in Cormen et al. [14, p. 16]. The input is a sequence of numbers $a_1, a_2, ..., a_n$. The output is a permutation of the input sequence of the form $a'_1, a'_2, ..., a'_n$, such that $a'_1 \leq a'_2 \leq ... \leq a'_n$. Note that the arrays are one-indexed in the pseudocode example, while they are zero-indexed in the corresponding Python and Java implementations.

```
insertion_sort(A):
  for i = 2 to A.length:
    key = A[i]
    // insert A[i] into sorted A[1..i-1]
    j = i - 1
    while j >= 1 and A[j] > key:
      A[j+1] = A[j]
      j      = j - 1
  A[j+1] = key
```

This relatively simple algorithm is sufficient to demonstrate several properties that would either need to be tested, or verified by the type checker, such as the property that the input is a sequence of numbers, that the output is a permutation of the input, and that the output is sorted.

The translation to Python is very close to the pseudocode just shown.

```
def insertion_sort(lst):
  for i in range(1, len(lst)):
    j   = i - 1
    key = lst[i]
    while j >= 0 and lst[j] > key:
      lst[j+1]  = lst[j]
      j        -= 1
    lst[j+1] = key
  return lst
```

[1] In 2006 Guido van Rossum, the creator of the Python programming language, gave a talk with the title "Design of Python" in the context of Stanford's CS242: Programming Languages, in which he refers to the absence of type checking in the compiler as the first of the "big ideas" of Python. One may question whether this indeed constitutes a notable novelty. The slides are available at: http://web.stanford.edu/class/cs242/slides/2006/python-vanRossum.ppt (accessed March 20, 2015).

Note that we are using Python 2 syntax, which would require minor modifications to make it valid Python 3. Lists in Python are untyped and may therefore contain an arbitrary collection of values. In addition to checking whether the output is indeed sorted and a permutation of the input, one would also have to ensure that the input list is a list of numbers. Otherwise, the results may not be as expected, as the following example shows.

```
insertion_sort([5, 4, 3, 2, 1, "foo", "bar"])
> [1, 2, 3, 4, 5, 'bar', 'foo']
```

One might question whether it is an example of the often-touted predictability of Python that strings are interpreted as being larger than any number. It is most certainly not self-evident, particularly for a programmer who is used to working in a language in which characters are internally represented by their ASCII value. Coming from such a background, the following output may be unexpected, since the uppercase letter A has the ASCII value 65.

```
insertion_sort(['a', 'A', 100, 101])
> [100, 101, 'A', 'a']
```

As this example demonstrates, the lack of typed lists may lead to a considerable amount of confusion that would be avoidable by using a programming language with a more expressive type system, such as Java. A possible implementation of the insertion sort algorithm in Java is given the following code listing.

```
public static int[] insertionSort(int[] arr) {
  for (int i = 1; i < arr.length; i++) {
    int key = arr[i];
    int j   = i - 1;
    while (j >= 0 && arr[j] > key) {
      arr[j+1] = arr[j];
      j        = j - 1;
    }
    arr[j+1]    = key;
  }
  return arr;
}
```

Unlike Python or some other dynamically typed programming language, Java utilises a type checker, which makes it possible to verify the input. Because lists are typed in Java, the type checker will reject input like in the Python examples given above. Thus, the entire class of test code that would have to be written to verify the legality of the input is no longer necessary. Verifying the output would still be necessary, however. There have been attempts to implement property-based testing in Java, similar to QuickCheck, which seem promising, even though a substantial amount of work is left to be done.[2]

Let us now move on to insertion sort in Haskell.

```
insertionSort :: Ord a => [a] -> [a]
insertionSort = foldr insert []

insert :: Ord a => a -> [a] -> [a]
insert x []     = [x]
insert x (y:ys)
  | x <= y    = x : y : ys
  | otherwise = y : insert x ys
```

Compared to Python or Java, this implementation of the insertion sort algorithm looks rather different. Note that there has been a minor change in the algorithm: the insertion starts at the beginning of the sorted list, while in the previous implementations the elements are moved downwards, starting from the back of the array. This change is due to the fact that a linked list is used as a data structure instead of an array. This does not negatively affect the runtime of the algorithm, however.

[2] At the time of writing, those Java implementations of QuickCheck lack several key features of the original. The most up-to-date implementation of property-based testing seems to be Paul Holsers' *junit-quickcheck*, which is available at: https://github.com/pholser/junit-quickcheck. It does not feature shrinking of test cases, nor does it generate increasingly complex test cases.

The Haskell code is more concise than the previously shown implementations. In fact, type annotations are optional, and could be omitted altogether due to Hindley-Milner type inference [21, 31]. In an example as short as the one given here, the advantages of Haskell may not be readily apparent. However, there is much greater flexibility with regards to the specification of the input. The Java code given above would have been slightly more verbose for polymorphic lists, but in Haskell it is trivial to generalise the code and present it in the form below. Even without type annotations, type inference would conclude that the input has to belong to the class Ord, i.e. the data type of the input has to belong to the type class of totally ordered data types.

While a Haskell programmer does not have to worry about mutable state, in this example, due to referential transparency, there is still the need to test whether certain properties of this algorithm hold. Unit testing is content with verifying examples, but property-based testing focuses on mathematical properties that are supposed to hold for all kinds of input. Finding such properties can be significantly more challenging than concocting a few examples for unit tests, though.

In the case of sorting, the properties can be taken from the pseudocode specification that was given above. To repeat, the desired properties are that, given a list of numbers as input, the output of the sorting function is a permutation of the input, and that the output is sorted. Remember that our implementation admits polymorphic lists of any ordered datatype as input. This leads to the following two properties, which can be easily verified by QuickCheck.

```
prop_sorted :: Ord a => [a] -> Bool
prop_sorted (x:y:z) = x <= y && prop_sorted (y:z)
prop_sorted _       = True

prop_permutation :: Eq a => [a] -> [a] -> Bool
prop_permutation xs ys = all null [xs \\ ys, ys \\ xs]
```

The property `prop_sorted` is relatively self-explanatory. A list of length two or more has to satisfy two criteria in order to be sorted. First, the head of the list has to be smaller than or equal to the head of the tail of the list. Second, the tail of the list has to be sorted as well. This recursion continues until the base case applies, according to which a list of length zero or one is sorted by definition.

The property `prop_permutation`, on the other hand, may be less obvious. Satisfying the permutation property is necessary since the output of a sorting function could sort the list that was provided as input, but drop some of the list values, for instance duplicate entries. In that case, the output would still be sorted, but it would only be a sorted subset of the input. Given a list of distinct elements, it should be obvious that the described property holds. However, the case of lists that contain duplicate entries warrants further discussion, as it builds upon knowledge of the implementation details of list subtraction in Haskell. The listing below illustrates an interaction with the Haskell interpreter GHCi.

```
> [1,1] \\ [1,1]
[]
> [1]   \\ [1,1]
[]
> [1,1] \\ [1]
[1]
```

Unsurprisingly, performing a list subtraction operation on two identical lists results in the empty list. Similarly, using a subtrahend that constitutes a superset of the minuend, like in the second example, results in the empty list. However, these examples alone would obfuscate the fact that list subtraction in Haskell takes individual elements into account. List subtraction could have been defined so that the third example given in the listing above returns an empty list as well. Alas, this was not what the Haskell implementers have done, which implies that the property `prop_permutation` is also valid for lists with duplicate entries.

Compared to manually specifying unit tests, having QuickCheck generate test cases for testing properties is arguably more elegant. Subjective aesthetic judgments put aside, testing properties by randomly generating test cases is also more reliable, due to the fact that the inputs used for testing are randomly generated. Consequently, they cover cases a programmer who writes unit tests one by one may not consider. From a methodological point of view, randomised testing does not provide absolute certainty, but for most practical purposes, successfully tested QuickCheck properties can be considered verified.

As the last sentence implies, there is a potential shortcoming of QuickCheck that can be alleviated by using a programming language with an even more advanced type system than Haskell's. Agda is an example of a programming language with dependent types, i.e. types that depend on values. This entails that it is possible to express a specification as part of the type signature, which has practical significance for software engineering as it drastically reduces the need to test software. Agda has further desirable qualities as well, which we are going to describe next.

2.5 Practical benefits of Agda for software development

Agda and similar programming languages like Coq [5] or Idris [8] make certified programming possible. The main idea is that certified programs supply their own proof of correctness. This is due to the Curry-Howard isomorphism, which relates computer programs and mathematical proofs [22]. Indeed, the previously mentioned programming languages double as proof assistants, which is possible due to the expressiveness of dependent types. *Programming* with dependent types is an area that is much less explored than constructing dependently-typed proofs, however.[3]

The goal of this section is to present a small number of Agda programs that illustrate how dependent types either eliminate entire classes of errors, or lead to greater expressivity. The latter is of practical importance for software engineering, provided one is willing to accept the claim that the ratio of software defects per lines of code is nearly constant, which implies that there are, in absolute terms, fewer software defects in more concise programming languages [29].

2.5.1 Avoiding out-of-bound errors

A prime example of the benefits of dependent types are length-indexed vectors. As mentioned above, in a language like C the abstraction of an array is not properly enforced by the type checker. Accessing the array location $A[m]$, where m is not within the boundaries of the array A, yields whatever value is stored in that particular memory location. In Python or Java this would lead to an exception. Even in Haskell accessing a location that does not exist leads to an exception.

Dependent types, however, eliminate the problem of out-of bound accesses and the resulting exceptions. The significance of this one problem cannot be overstated. Tony Hoare, who discovered QuickSort, developed Hoare logic, and defined the formal language Communicating Sequential Processes (CSP), also invented the *null reference* when designing ALGOL W. In 2009 Hoare called the null reference his "billion dollar mistake".[4] Agda deals with the problem of null references problem in a rather elegant manner.

Let us define a lookup function that takes as arguments a polymorphic vector as well as a Peano number[5] from a finite set of natural numbers. The latter indicates the position of the element in the vector that is supposed to be retrieved. The Vec datatype is a length-indexed vector of length n, while Fin is a non-negative integer i, such that $0 \leq i < n$.

```
_!_ : {A : Set} {n : Nat} -> Vec A n -> Fin n -> A
(x :: xs) ! fzero   = x
(x :: xs) ! fsucc i = xs ! i
```

This function traverses a vector via recursion, until the desired element is reached. Remarkably, it is impossible to call this function with an argument of type Fin that is greater than the length of the vector, as this indicates an impossible case. Such programs would be rejected during type checking, which consequently precludes out-of-bound errors.

2.5.2 Termination checking

Agda's termination checker, which is based on Abel's *foetus* termination checker [1], deserves its own section. In Agda, all functions are total functions. In mathematics, this term stands for functions that are defined for all possible input values. The interpretation in functional programming is that a total function has to terminate for all possible input values. Note that functional purity implies that, given a particular input value, a function always returns the same result.

Of course, due to the undecidability of the halting problem, termination checking is not solvable in full generality. This does not imply that termination checking is generally impossible, though. The key element is the presence of structurally recursive functions, i.e. recursive functions that consume their arguments, for instance by processing a list element by element or manipulating a given numerical argument so that it decreases with each recursive call. On the other hand, termination fails in the presence of what Felleisen et. al [19] refer to as generative recursive functions, i.e. recursive functions that generate a new piece of data for subsequent recursive calls. For the sake of completeness, the pathological case of a recursive function that neither consumes its input, nor generates new data should be included as well. Examples for those three categories of recursion are given below.

Arguably the most well-known recursive function is related to the computation of Fibonacci numbers. It is easy to see that the Fibonacci function eventually terminates, due to consumption of the input value, which makes it a case of a structurally recursive function.

```
fib : Nat -> Nat
fib zero           = zero
fib (suc zero)     = suc zero
fib (suc (suc n)) = fib (suc n) + fib n
```

A famous example of a generative recursive function is the computation of the Collatz conjecture, but this would lead to a somewhat more complicated Agda program. Thus, a simpler example has to suffice, which passes the type checker but is rejected by the termination checker.

[3] At the time of writing, Chlipala's book *Certified Programming with Dependent Types* [9] seems to be the only current source on that topic that is easily accessible outside academia.

[4] At the software development conference QCon London 2009, Tony Hoare gave a talk with the title "Null References: The Billion Dollar Mistake". A recording as well as the slides of the presentation are available at: http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare (accessed March 30, 2015).

[5] Peano numbers are constructed from values indicating either zero or the successor of a number. For instance, 0 is represented as $zero$, 1 as $succ(zero)$, 2 as $succ(succ(zero))$, and so on. The construction of Church numerals, which we will encounter shortly, is similar.

```
infiniteLoop : Nat -> Nat
infiniteLoop n = infiniteLoop (suc n)
```

Note that new data is generated from the input value. More concretely, the given non-negative integer is incremented by 1 in each iteration. It is trivial to turn this example into a pathological case of a function that neither consumes its input nor generates new data from the given input for further recursive calls, by simply passing on the argument to the recursive call without modification.

```
infiniteLoop : Nat -> Nat
infiniteLoop n = infiniteLoop n
```

In earlier versions of Agda, such functions were highlighted when using the Agda mode in the Emacs text editor, but those programs could still be executed. For the same effect in current versions of Agda, one would have to use so-called language pragmas, i.e. special commands that are passed on to the Agda interpreter or compiler. One such pragma, `NON_TERMINATING`, relaxes the requirement that all functions have to be total and need to pass the termination checker. This is partly a concession to the halting problem, since there are seemingly sound programs, such as a program that computes the Collatz conjecture, that are believed to terminate for all legal inputs, even though a proof is still missing.

2.5.3 Specifications as types

Lastly, the type system of Agda is able to express propositions as types, which makes it possible to encode specifications in the type signature. One of the simplest examples with a practical application is the concatenation of two vectors. This operation can be defined as follows.

```
_++_ : forall {m n} {A : Set} ->
       Vec A m -> Vec A n -> Vec A (m + n)
[]         ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

The resulting type is a vector of the combined length of both input vectors. Indeed, the length of the vector that constitutes the result of this function is computed at the type level. This is far from trivial, as this ensures that oversights by the programmer are effectively caught. For instance, the following definition is rejected by the Agda type checker because the resulting vector is not of the specified length.

```
_++_ : forall {m n} {A : Set} ->
       Vec A m -> Vec A n -> Vec A (m + n)
[]         ++ ys = ys
(x :: xs) ++ ys = (xs ++ ys)
```

This is a simple example chosen for illustrative purposes. Nonetheless, it shows that entire classes of errors are caught by the Agda type checker that would require testing in a programming language with a less expressive type system.

3. Parts of the λ-calculus

3.1 Preliminaries

The λ-calculus constitutes a model of computation. It was initially introduced by Church in the 1930s in the context of a paper on the decision problem [10]. In the 1950s, McCarthy used the λ-calculus as the basis of the functional programming language Lisp [28], which is syntactically rather close to typical presentations of the λ-calculus. While the syntax of functional programming languages has changed significantly, with OCaml or Haskell looking rather different from Lisp, they nonetheless ultimately share the same theoretical foundation.

The theory that underlies the λ-calculus is rather complex, and therefore cannot be presented in its entirety. Consequently, this section merely attempts to provide an overview of some variants of the λ-calculus inasmuch as they are relevant for later sections of this paper. The goal is to illustrate the basic concepts behind the simplest form of λ-calculus, the untyped λ-calculus, as well as several extensions. Please note that in the following we are describing call-by-value variants of the various λ-calculi.

The presentation below mainly follows Pierce [37], but also Thompson [43], who both discuss the λ-calculus in the context of type systems. A much more thorough treatment of the (untyped) λ-calculus is provided in a classic monograph by Barendregt [4]. A conceptual framework of the various λ-calculi and how they relate to each other is provided by Bardendregt's λ-cube [3], which we will briefly discuss much further below.

3.2 The untyped λ-calculus (λ)

3.2.1 Basic elements

The simplest form of the λ-calculus is the untyped λ-calculus. In fact, it is so simple that a novice may even question whether it is useful as a model of computation. Some practical examples are therefore in order to illustrate its suitability. Those examples also show that the untyped λ-calculus is not the most practical foundation for a programming language.

In a nutshell, the untyped λ-calculus consists of λ-terms that are inductively defined. Due to being defined inductively, terms can be nested arbitrarily deeply. A λ-term can either be a variable, an application, or an abstraction. The grammar of the untyped λ-calculus [37, p. 72] is as follows.

$$
\begin{array}{llr}
t ::= & & \text{terms:} \\
& x & \text{variable} \\
& \lambda x\, .\, t & \text{abstraction} \\
& t\; t & \text{application} \\
\\
v ::= & & \text{values:} \\
& \lambda x\, .\, t & \text{abstraction value}
\end{array}
$$

The structural operational semantics for the evaluation of t to get the result t', the relation $t \rightarrow t'$, are as follows [37, p. 72].

$$\frac{t_1 \rightarrow t_1'}{t_1\; t_2 \rightarrow t_1'\; t_2} \qquad (E_{\mathit{app\text{-}1}})$$

$$\frac{t \rightarrow t'}{v\; t \rightarrow v\; t'} \qquad (E_{\mathit{app\text{-}2}})$$

$$(\lambda x\, .\, t)\; v \rightarrow [x \mapsto v]t \qquad (E_{\mathit{beta}})$$

To give a very simple example: the identity function can be defined as $\lambda x\, .\, x$. Slightly more involved is the example of a function that returns the sum of its arguments. Further below we will show how to define numbers in the untyped λ-calculus, but for now we take them as given. Furthermore postulating a binary function $+$ that adds its arguments, one possible corresponding λ-expression is $\lambda x\, .\, (\lambda y\, .\, +\; x\; y)$. In case we wanted to add the numbers 2 and 4, the evaluation is as follows.

$$
\begin{aligned}
& ((\lambda x\, .\, \lambda y\, .\; +\; x\; y)\; 4)\; 2 \\
&= (\lambda y\, .\, +\, 2\; y)\; 4 \\
&= \;+\; 2\; 4 \\
&= 6
\end{aligned}
$$

So-called Polish prefix notation is customary, and furthermore illustrates the fact that the λ-calculus exclusively uses functions. Thus, $+$ is not an operator but a function that takes two arguments. Using infix notation instead of prefix notation would have obscured this fact. Admittedly, it can take some time to get used to Polish prefix notation. The skeptic, however, may want to recall that (reverse) Polish notation has turned out to be rather useful for the efficient implementation of stack-based programming languages.

In the λ-calculus, all functions only take one argument. As the previous example has illustrated, functions can easily be applied to functions, which is practically identical to having a single function that takes multiple arguments. It is therefore conventional to write, for instance, $\lambda x\ y\ .\ +\ x\ y$ instead of $\lambda x\ .\ \lambda y\ .\ +\ x\ y$.

An important distinction in the λ-calculus is between *free variables* and *bound variables*. Bound variables are within the scope of a λ-abstraction, while free variables are not. For instance, take the λ-expression $\lambda y\ .\ +x\ y$. In this example, x is a free variable, while y is a bound variable.

3.2.2 Reduction rules

λ-expressions are evaluated by the application of three reduction rules, which are α-conversion, β-reduction, and η-conversion. α-conversion is the process of renaming bound variables, for instance $\lambda x\ .\ x$ to $\lambda y\ .\ y$. Not all instances of α-conversion are that trivial, however. The main practical difficulty is the problem of variable capture, which has to be avoided. While it is correct, if potentially confusing, to rewrite $\lambda x\ .\ \lambda y\ .\ y$ as $\lambda x\ .\ \lambda x\ .\ x$, it would be wrong to rewrite $\lambda y\ .\ \lambda x\ .\ y$ as $\lambda x\ .\ \lambda x\ .\ x$. The problem of variable capture can be avoided in a rather straightforward manner by restricting the result of an α-conversion to variable names that have not been used yet.

We have already seen an example of β-reduction above, namely in the application of arguments to the addition function. More formally, β-reduction is the substitution of all free variables by the provided term. This means that $(\lambda x\ .\ e)\ e'$ results in $[x := e']e$. A simple illustrative example is given below.

$$\begin{aligned} &(\lambda y\ .\ +x\ y)\ 2 \\ =\ &+x\ 2 \end{aligned}$$

However, any free variables in e' may not be captured by a λ-abstraction in e. In order to avoid variable capture, it is necessary to rename the affected variable before applying y.

$$\begin{aligned} &(\lambda x\ .\ \lambda y\ .\ x)\ y \\ =\ &(\lambda x\ .\ \lambda y'\ .\ x)\ y \\ =\ &\lambda y'\ .\ y \end{aligned}$$

The function we have just seen returns a constant. Without renaming, though, the result is variable capture, which turns this function into the identity function. This would not be a valid reduction in this case.

$$\begin{aligned} &(\lambda x\ .\ \lambda y\ .\ x)\ y \\ \neq\ &\lambda y\ .\ y \end{aligned}$$

Lastly, there is η-conversion, which can be justified by referring to the concept of extensional equality of functions, as it is understood in mathematics.[6] This means that two functions exhibit the property of extensional equality if they produce the same output for all possible inputs. Concretely, this means that in the λ-calculus it is possible to rewrite λ-expressions by removing redundant λ-abstractions since doing so would not change the evaluation. The rewriting rule can be expressed as follows. A necessary condition is that the variable x may not appear as a free variable anywhere in f.

$$\begin{aligned} &(\lambda x\ .\ fx) \\ =\ &f \end{aligned}$$

Given those three reduction rules, one might ask whether it matters in which order they are applied. An important theoretical result in that regard is the Church-Rosser theorem [11], according to which each λ-expression has exactly one normal form that is reachable through β-reduction or $\beta\eta$-reduction. α-conversion is excluded since it is non-terminating, because variables can be renamed *ad infinitum*.

A λ-expression is said to be in normal form if it cannot be β-reduced or $\beta\eta$-reduced any further. Church-Rosser have proven that the order of the application of the reduction rules does in fact not matter. On a side note, this theoretical finding has far-reaching practical consequences, as it means that λ-expressions can not only be evaluated in any order, but in parallel as well, which is relevant for the current engineering challenge of exploiting the power of multi-core and many-core processors.

3.2.3 Additions to the untyped λ-calculus

The untyped λ-calculus as presented above is rather sparse. It can be enriched with boolean values, natural numbers, and a combinator for recursion, which make the untyped λ-calculus more usable for modelling computations.

Since we are only dealing with functions, we will have to use functions to encode data types, for instance booleans. Speaking of boolean values, the value $true$ is conventionally represented as $\lambda x\ y\ .\ x$, which is a function that returns its first argument, ignoring its second argument. Conversely, the boolean value $false$ is represented as $\lambda x\ y\ .\ y$, which is a function that returns its second argument, ignoring its first argument.

Functions also have to be used to represent natural numbers. In the encoding chosen by Church, commonly referred to as Church numerals, a natural number n is a higher-order function that is applied to its argument n times. To use the possibly more readable notation from mathematics:

$$\begin{aligned} \underline{0}fx &= x \\ \underline{1}fx &= fx \\ \underline{2}fx &= f(fx) \\ &\dots \end{aligned}$$

The untyped λ-calculus is expressive enough to encode *combinators*, i.e. higher-order functions that define functions without variables. The most important one is the Y combinator [43, p.41–

[6] η-conversion is covered in this section because it is relevant for the λ-calculus. It is irrelevant for any of the rules we later on specify. As Andreas Abel pointed out to me, η-conversion is redundant for the implementation of functional programming languages, but is potentially useful in non-lazy functional programming languages like Standard ML or Scheme. In such programming languages, manual η-expansion, i.e. wrapping a λ-term in a redundant λ-abstraction, prevents immediate evaluation. η-conversion is irrelevant for our work, however, since the presence or absence of η-conversion will not affect the actual translation.
Daniel Lee remarked that η-conversion is very important in some domains. For instance, so-called η-long normal form provides a more useful normal form when dealing with logics from a proof-theoretic perspective [17]. Further, η-long normal form is required by Watkin's hereditary substitution [15, 45], which is a substitution model that automatically contracts secondary redices.

42], which, perhaps surprisingly, makes it possible to encode recursion in the λ-calculus, as the following equations illustrate.

$$\begin{aligned}Y\ g &= (\lambda f\ .\ (\lambda x\ .\ f\ (x\ x))\ (\lambda x\ .\ f\ (x\ x)))\ g\\ &= (\lambda x\ .\ g\ (x\ x))\ (\lambda x\ .\ g\ (x\ x))\\ &= g\ ((\lambda x\ .\ g\ (x\ x))\ (\lambda x\ .\ g\ (x\ x)))\\ &= g\ (Y\ g)\end{aligned}$$

The equation $F\ g = g\ (F\ g)$ is not a mathematical absurdity, but instead implies that those evaluations can be repeated infinitely many times. This is illustrated by the equations below.

$$\begin{aligned}Y\ g &= g\ (Y\ g)\\ &= g\ (g\ (Y\ g))\\ &= g\ (g\ (g\ (Y\ g)))\\ &= \ldots\end{aligned}$$

3.3 The simply typed λ-calculus ($\lambda_{\to}$)

If we add simple data types like natural numbers or booleans to the untyped λ-calculus, it is possible to write λ-expressions that will eventually get stuck during evaluation. Given an *if-then-else* construct and evaluation rules according to which natural numbers do not double as boolean values — for instance, in the programming language C, the integer 0 is interpreted as the boolean value *false*, while any non-zero integer is interpreted as the boolean value *true* —, the term `if 0 then x else y` cannot be evaluated [37, p. 99]. The addition of typing rules solves this problem, as they would reject an expression like the preceding one as illegal.

The definition of the simply typed λ-calculus, syntax, structural operational semantics, and typing rules, are given below, following Pierce [37, p. 103].

$t ::=$		terms:
	x	variable
	$\lambda x : T\ .\ t$	abstraction
	$t\ t$	application
$v ::=$		values:
	$\lambda x : T\ .\ t$	abstraction value
$T ::=$		types:
	$T \to T$	function type
$\Gamma ::=$		contexts:
	$\varnothing$	empty context
	$\Gamma, x : T$	context, term variable binding

Compared to the untyped λ-calculus, this grammar is extended to account for a type in λ-abstractions, signified by the letter T. Further, a context, signified by the letter Γ is added, which is necessary for looking up the types of variables during type checking. Note that the context is inductively defined.

The structural operational semantics for the relation $t \to t'$ are largely unchanged from the untyped λ-calculus. The only modification is the addition of a type to the argument in the rule E_{beta}.

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2} \qquad (E_{app\text{-}1})$$

$$\frac{t \to t'}{v\ t \to v\ t'} \qquad (E_{app\text{-}2})$$

$$(\lambda x : T\ .\ t)\ v \to [x \mapsto v]t \qquad (E_{beta})$$

The typing rules for $\Gamma \vdash t : T$ are an addition to the previously presented simply typed λ-calculus.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad (T_{var})$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1\ .\ t : T_1 \to T_2} \qquad (T_{abs})$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2} \qquad (T_{app})$$

According to the typing rule T_{var}, given a context Γ and a variable x, the variable x is assigned to type T in the context Γ. The rule T_{abs} specifies that, given a λ-abstraction, the resulting function type maps type T_1 of its argument to T_2, which is the type of the λ-term t. The λ-term t is assigned to type T_2 under the assumption that $\Gamma, x : T_1$. Lastly, the typing rule T_{app} describes that term t_1 of a λ-application has to be of a function type whose domain is identical to the type of the term t_2. Consequently, the resulting type is T_2, since T_1 is the type of the argument of a function from T_1 to T_2.

3.4 The polymorphic λ-calculus (System F)

The polymorphic λ-calculus, which is also referred to as the second-order λ-calculus, adds universal quantification over types to the simply typed λ-calculus. It was independently discovered in the 1970s by Girard [20] and Reynolds [39]. The polymorphic λ-calculus is of great significance as it provides the theoretical foundation for functional programming languages like Haskell, or ML and its many dialects.

The benefit of type-polymorphism is easy to see. To condense the presentation, we will use Haskell for the following illustration. Imagine you wanted to moderately exercise your programming skills and write a function that, given a value, produces a singleton list with that value. Unfortunately, though, you are restricted to explicitly typed functions. To start with, you would have to cover booleans and integers separately.

```
makeSingletonBool :: Bool -> [Bool]
makeSingletonBool b = [b]

makeSingletonInteger :: Integer -> [Integer]
makeSingletonInteger i = [i]
```

It gets worse from there, however. Haskell many more data types: *Char*, *String*, *Float*, *Double*, *Int*, and so on. In addition, it is possible to define new data types in Haskell, which means that, without type polymorphism, a potentially infinite number of `makeSingleton` functions would have to be written, to take all those types into account. Understandably, this would be rather tedious, to say the least.

Fortunately, Haskell is based on System F. To be more precise, Haskell is based on System F_C(X)[42], which is a superset of System F. This implies that we can make use of type polymorphism. Consequently, a `makeSingleton` function that accepts arguments of any type as input, due to parametric polymorphism, can be expressed in the following way.

```
makeSingleton :: a -> [a]
makeSingleton x = [x]
```

This means that whatever the type of the value that is given as an argument to this function is, the result will be a singleton list that contains this value. The practical consequence is that this one function definition that makes use of type polymorphism replaces a potentially infinite number of function definitions that would only cover one type each.

When using the Haskell compiler GHC, the language pragma `ExplicitForAll` can be used to make the quantification over type parameters explicit:

```
makeSingleton :: forall a . a -> [a]
makeSingleton x = [x]
```

In short, System F extends the simply typed λ-calculus with type polymorphism. While this feature makes life for the programmer more convenient, it leads to a significantly more complex underlying representation.

The syntax, structural operational semantics, and typing rules of System F are given below, following Pierce [37, p. 343]. Overall, System F is a rather straightforward extension of the previously described simply typed λ-calculus. Additions are terms for type abstraction and type application, a type abstraction value, a type variable X that encodes parametric polymorphism, as well as a universal type, which we have just seen expressed in Haskell. The context Γ is extended to take type variables into account as well. This leads to the following specification of the syntax of System F.

$t ::=$		terms:
	x	variable
	$\lambda x : T \,.\, t$	abstraction
	$t\ t$	application
	$\lambda X \,.\, t$	type abstraction
	$t[T]$	type application
$v ::=$		values:
	$\lambda x : T \,.\, t$	abstraction value
	$\lambda X \,.\, t$	type abstraction value
$T ::=$		types:
	X	type variable
	$T \to T$	function type
	$\forall X \,.\, T$	universal type
$\Gamma ::=$		contexts:
	$\varnothing$	empty context
	$\Gamma, x : T$	context, term variable binding
	Γ, X	context, type variable binding

Two rules for type application were added to the structural operational semantics for the relation $t \to t'$:

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2} \qquad (E_{app\text{-}1})$$

$$\frac{t \to t'}{v\ t \to v\ t'} \qquad (E_{app\text{-}2})$$

$$(\lambda x : T \,.\, t)\ v \to [x \mapsto v]t \qquad (E_{beta})$$

$$\frac{t \to t'}{t[T] \to t'[T]} \qquad (E_{t\text{-}app})$$

$$(\lambda X \,.\, t)[T] \to [X \mapsto T]t \qquad (E_{t\text{-}beta})$$

The typing rules for the judgment $\Gamma \vdash t : T$ are:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad (T_{var})$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1 \,.\, t : T_1 \to T_2} \qquad (T_{abs})$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2} \qquad (T_{app})$$

$$\frac{\Gamma, X \vdash t : T}{\Gamma \vdash \lambda X \,.\, t : \forall X \,.\, T} \qquad (T_{t\text{-}abs})$$

$$\frac{\Gamma \vdash t : \forall X \,.\, T_1}{\Gamma \vdash t[T_2] : [X \mapsto T_2]T_1} \qquad (T_{t\text{-}app})$$

3.5 The simply typed λ-calculus with type operators (λ_ω)

Compared to the simply typed λ-calculus, the simply typed λ-calculus with type operators adds, unsurprisingly, type operators. Concretely, this means that this variant of the λ-calculus enables us to use abstraction and application at the type level. Please note that the simply typed λ-calculus with type operators does not extend System F, but the simply typed λ-calculus.

The following λ-expression illustrates how to apply a type to the polymorphic identity function $\lambda X \,.\, \lambda x : X \,.\, x$. Again, this is hardly a complex example. It is only intended to illustrate type application. Type abstraction is indicated by λX in the example below, which indicates that any type in the simply typed λ-calculus can be applied to it.

$$\begin{aligned} &(\lambda X \,.\, \lambda x : X \,.\, x)\ \mathit{Bool} && (\text{applying } \mathit{Bool}) \\ &= \lambda x : \mathit{Bool} \,.\, x \end{aligned}$$

Thus, the application of the type Bool to the given function results in an identity function for arguments of type Bool. Since X is merely a placeholder for some type, any type could be applied to that function to yield an identity function that is restricted to that particular type.

Because abstraction and application are available at the type level, identical types can be expressed in multiple ways. In fact, the same type can be expressed in potentially infinitely many ways. The definition of the simply typed λ-calculus with type operators therefore requires rules for type equivalence. An example of type equivalence, following Pierce [37, p. 441] is given below. Assume the existence of Id as a synonym of the type operator $\lambda X \,.\, X$. Consequently, the expression $\mathbb{N} \to \mathit{Bool}$ can be expressed in potentially infinitely many ways:

$$\begin{aligned} &\mathbb{N} \to \mathit{Bool} \\ &= \mathit{Id}\ \mathbb{N} \to \mathit{Bool} \\ &= \mathbb{N} \to \mathit{Id}\ \mathit{Bool} \\ &= \mathit{Id}\ (\mathbb{N} \to \mathit{Bool}) \\ &= (\mathit{Id}\ \mathbb{N}) \to (\mathit{Id}\ \mathit{Bool}) \\ &= \ldots \end{aligned}$$

A further addition of the simply typed λ-calculus with type operators are *kinds* [37, p. 441]. Kinds are the types of types, and a means of classifying type expressions based on their arity, not too dissimilar to regular function types. However, the difference is that kinds describe the structure of the types of a function. We will encounter some examples in a short while, which illustrate that the arity of the kinds is not necessarily identical to the arity of the function whose structure of types they describe.

Kinds are inductively defined, having either a proper type, which is expressed by the symbol $*$, or a compound expression whose parts are connected with the symbol $\Rightarrow$. Examples of proper types are *Bool*, $\mathbb{N}$, $\mathbb{N} \rightarrow \mathbb{N}$, and so on. To give another example, a function that takes any two arguments belonging to the same type for which equality is definable, and returns a *Bool*, i.e. the polymorphic equality function, has the kind $* \Rightarrow *$, while a monomorphic equality functions for integers has kind $*$. To round out this example: if we felt particularly creative and defined such an equality function to take any two arguments for which equality is definable, but to return any kind of result, the corresponding kind would be $* \Rightarrow * \Rightarrow *$.

The full specification of the simply typed λ-calculus with type operators is given below, following Pierce [37, p. 446]. The grammar below extends the simply typed λ-calculus. Kinds were already discussed. Other additions include type variables, operator abstraction as well as operator application. Furthermore, the rules for the context Γ were expanded to add type variable bindings.

$t ::=$		terms:
	x	variable
	$\lambda x : T \,.\, t$	abstraction
	$t\ t$	application
$v ::=$		values:
	$\lambda x : T \,.\, t$	abstraction value
$T ::=$		types:
	X	type variable
	$\lambda X :: K \,.\, T$	operator abstraction
	$T\ T$	operator application
	$T \rightarrow T$	function type
$\kappa ::=$		kinds:
	$*$	kind of proper types
	$\kappa \Rightarrow \kappa$	kind of operators
$\Gamma ::=$		contexts:
	$\varnothing$	empty context
	$\Gamma, x : T$	context, term variable binding
	$\Gamma, X :: \kappa$	context, type variable binding

Compared to the simply typed λ-calculus, the structural operational semantics for the relation $t \rightarrow t'$ are unchanged:

$$\frac{t_1 \rightarrow t_1'}{t_1\ t_2 \rightarrow t_1'\ t_2} \qquad (E_{app\text{-}1})$$

$$\frac{t \rightarrow t'}{v\ t \rightarrow v\ t'} \qquad (E_{app\text{-}2})$$

$$(\lambda x : T \,.\, t)\ v \rightarrow [x \mapsto v]t \qquad (E_{t\text{-}beta})$$

The kinding rules for the judgment $\Gamma \vdash T :: \kappa$ are relatively straightforward and conceptually fairly similar to the typing rules we have seen so far.

$$\frac{X :: \kappa \in \Gamma}{\Gamma \vdash X :: \kappa} \qquad (\kappa_{var})$$

$$\frac{\Gamma, X :: \kappa_1 \vdash T :: \kappa_2}{\Gamma \vdash \lambda X :: \kappa_1 \,.\, T :: \kappa_1 \Rightarrow \kappa_2} \qquad (\kappa_{abs})$$

$$\frac{\Gamma \vdash T_1 :: \kappa \Rightarrow \kappa' \quad \Gamma \vdash T_2 :: \kappa}{\Gamma \vdash T_1\ T_2 :: \kappa'} \qquad (\kappa_{app})$$

$$\frac{\Gamma \vdash T_1 :: * \quad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \rightarrow T_2 :: *} \qquad (\kappa_{arr})$$

Type equivalence was illustrated above. The following set of type equivalence rules gives a formal specification for the relation $S \equiv T$. These rules are required by the rule $T_{t\text{-}eq}$, which is mentioned as part of the typing rules further below.

$$T \equiv T \qquad (Q_{refl})$$

$$\frac{T \equiv S}{S \equiv T} \qquad (Q_{symm})$$

$$\frac{S \equiv U \quad U \equiv T}{S \equiv T} \qquad (Q_{trans})$$

$$\frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 \rightarrow S_2 \equiv T_1 \rightarrow T_2} \qquad (Q_{arr})$$

$$\frac{S \equiv T}{\lambda X :: \kappa \,.\, S \equiv \lambda X :: \kappa \,.\, T} \qquad (Q_{abs})$$

$$\frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1\ S_2 \equiv T_1\ T_2} \qquad (Q_{app})$$

$$(\lambda X :: \kappa \,.\, T_1)\ T_2 \equiv [X \mapsto T_2]\ T_1 \qquad (Q_{beta})$$

Lastly, the typing rules are mostly familiar, with the main change being the addition of the rule $T_{t\text{-}eq}$, which makes use of the type equivalence rules that were just specified.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad (T_{var})$$

$$\frac{\Gamma \vdash T_1 :: * \quad \Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1 \,.\, t : T_1 \rightarrow T_2} \qquad (T_{abs})$$

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2} \qquad (T_{app})$$

$$\frac{\Gamma \vdash t : S \quad S \equiv T \quad \Gamma \vdash T :: *}{\Gamma \vdash t : T} \qquad (T_{t\text{-}eq})$$

When combined with System F, the simply typed λ-calculus with type operators forms System F$_\omega$. Together with the specification of the programming language Agda, System F_ω is part of the next section.

4. Translating Agda to System $\mathbf{F}_\omega$

4.1 Theoretical background

So far we have discussed the various λ-calculi as separate entities. As the successive presentation of more and more complex λ-calculi may suggest, though, it is possible to translate a computation that is expressed in one particular kind of λ-calculus into a different kind of λ-calculus. Bardendregt's λ-cube [3] illustrates the relations between the *typed* λ-calculi we have seen so far, in addition to several others. The λ-cube is reproduced below, with the shorthands we have been using in this paper.[7]

$$\begin{tikzcd}
 & F_\omega \arrow[rr] & & FP_\omega \\
F \arrow[ur] \arrow[rr] & & FP \arrow[ur] & \\
 & \lambda_\omega \arrow[uu] \arrow[rr] & & \lambda P_\omega \arrow[uu] \\
\lambda_\to \arrow[uu] \arrow[ur] \arrow[rr] & & \lambda P \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

Figure 1. Barendregt's λ-cube

We have discussed several typed λ-calculi. To locate them in the λ-cube: the simply typed λ-calculus is represented as $\lambda_\to$. We have seen that $\lambda_\to$ can easily be extended into the simply typed λ-calculus with type operators or the second-order or polymorphic λ-calculus, also referred to as System F, abbreviated as F on the λ-cube. System F and the simply typed λ-calculus with type operators can be combined to create System F_ω, abbreviated as F_ω on the λ-cube. We will discuss System F_ω in the next subsection.

So far, we have covered the left side of the λ-cube, and quickly traced the relations between four typed λ-calculi, which mirrors our previous discussion. The right side of the λ-cube uses suspiciously similar shorthands. The letter P is a shorthand for dependent types. As the λ-cube shows, each of the four typed λ-calculi we have discussed can be enriched by adding dependent types to them. The type system that is used by Agda is closely related to λP_ω. We will shortly discuss this relation in greater detail.

The key insights of the λ-cube are the relationships between the various typed λ-calculi [3, p. 5]. All eight of the typed λ-calculi that are represented in the λ-cube allow terms that depend on terms, which enables monomorphic types. The four λ-calculi on the top face of the λ-cube allow terms that depend on types, which enables polymorphism. Note that these four λ-calculi permit impredicativity, i.e. self-referencing definitions, which is prohibited in Agda. The four λ-calculi on the right face of the λ-cube allow types that depend on terms, i.e. dependent types. Lastly, the four λ-calculi on the back face of the λ-cube allow types that depend on types, which leads to type constructors. Note that FP_ω is the most expressive λ-calculus on the λ-cube. We have chosen this shorthand for consistency, even though this λ-calculus is often referred to as λC, as it is also known as Coquand's *Calculus of Constructions* [13], a very well-studied λ-calculus that forms the basis of the proof assistant Coq.

The previous relations have been expressed in more formal terms as well. Barendregt uses a notation that follows the pattern $PTS = \langle S, A, R\rangle$, which, of course, warrants further explanations. Concretely, this notation expresses that a pure type system (PTS) consists of a triple that specifies a PTS by giving its set of sorts (S), set of axioms (A), and set of rules (R). The sorts are types, denoted by $*$, or kinds, denoted by $\Box$; set A is defined as $* : \Box$, indicating that the sort of types is a kind. R then describes the sort of the dependent product type that is available in the given λ-calculus. Slightly modifying Barendregt's notation, the corresponding rule, provided $(s_1, s_2, s_3) \in R$, is:

$$\frac{\Gamma \vdash U : s_1 \quad \Gamma, x : U \vdash T : s_2}{\Gamma \vdash (x : U) \to T : s_3} \tag{Π}$$

Since s_3 is consequently identical to s_2, s_3 can be omitted when classifying the λ-calculi in the λ-cube. Thus, λ-calculi with terms dependent on types, the top face in the λ-cube, are specified as $(\Box, *)$. λ-calculi with types dependent on types, the back face on the λ-cube, as $(\Box, \Box)$, and λ-calculi with types dependent on terms, the right face of the λ-cube, as $(*, \Box)$. Lastly, λ-calculi where terms depend on terms, which applies to all eight λ-calculi in the λ-cube, are specified as $(*, *)$.

Agda does not directly relate to any of the eight λ-calculi of the λ-cube. However, λP_ω corresponds to Agda without universes. If one wanted to locate Agda on the λ-cube, its location would be somewhere between λP_ω and FP_ω. Consequently, the task of translating Agda to System F_ω can be expressed as a relation from λP_ω, with the addition of universes, to F_ω. To visualise this, pick a point on the relation arrow from λP_ω to FP_ω, and draw a relation arrow from that point to F_ω. It may not be intuitively clear why this translation is viable. While there are some Agda programs that have no direct equivalent in System F_ω, type coercions can be used to make them expressible in that particular λ-calculus. In our extended specification of System F_ω, which is presented further below, this case corresponds to an unknown type, expressed as Any.

Agda is not Turing complete as it requires all programs to terminate. This restriction does apply to System F_ω as well, but not to Haskell. As a consequence, there is a class of Haskell programs, the class of non-terminating programs, that is legal in Haskell, but illegal in System F_ω as well as Agda. It would be impossible to find an equivalent legal Agda program for a non-terminating legal Haskell program, while the opposite is feasible. The latter is the problem of full abstraction, which was first studied by Milner as well as Plotkin [30, 38]. Concretely, the task of a compiler that performs a source-to-source translation from Agda to System F_ω and, eventually, to Haskell is to find an equivalent Haskell program that exhibits the same behaviour as the Agda program that was provided as input. This implies that we are assuming an extensional interpretation of equivalence.

4.2 The higher-order polymorphic lambda-calculus (System $\mathbf{F}_\omega$)

The higher-order polymorphic lambda-calculus (System F_ω) combines the polymorphic λ-calculus (System F) with the simply typed λ-calculus with type operators (λ_ω). The formal specification, which is presented below, is largely identical to both the definitions of System F and the simply typed λ-calculus with type operators, with the exception of *kinding annotations*. Those kinding annotations appear in type abstractions and quantifiers or, expressed more abstractly, in positions where type variables are bound [37, p. 449–451].

[7] There is a mismatch between our notation, which is taken from Pierce, and Barendregt's notation. Pierce abbreviates the simply typed λ-calculus with type operators as λ_ω, while Barendregt uses the shorthand $\lambda_{\underline{\omega}}$. System F, the polymorphic λ-calculus is λ_2 in Barendregt's notation. Furthermore, Barendregt uses the shorthand λ_ω for System F_ω.

The syntax of System F$_\omega$ is as follows:

$t ::=$		terms:
	x	variable
	$\lambda x : T \,.\, t$	abstraction
	$t\ t$	application
	$\lambda X :: \kappa \,.\, t$	type abstraction
	$t\ [T]$	type application
$v ::=$		values:
	$\lambda x : T \,.\, t$	abstraction value
	$\lambda X :: \kappa \,.\, t$	type abstraction value
$T ::=$		types:
	X	type variable
	$T \to T$	function type
	$\forall X :: \kappa \,.\, T$	universal type
	$\lambda X :: \kappa \,.\, T$	operator abstraction
	$T\ T$	operator application
$\Gamma ::=$		contexts:
	$\varnothing$	empty context
	$\Gamma, x : T$	context, term variable binding
	$\Gamma, X :: \kappa$	context, type variable binding
$\kappa ::=$		kinds:
	$*$	kind of proper types
	$\kappa \Rightarrow \kappa$	kind of operators

The structural operational semantics for the relation $t \to t'$ are as follows. Compared to the λ-calculi we have discussed before, the changes are minor.

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2} \qquad (E_{\textit{app-1}})$$

$$\frac{t \to t'}{v\ t \to v\ t'} \qquad (E_{\textit{app-2}})$$

$$(\lambda x : T \,.\, t)\ v \to [x \mapsto v]t \qquad (E_{\textit{beta}})$$

$$\frac{t \to t'}{t\ [T] \to t'[T]} \qquad (E_{\textit{t-app}})$$

$$(\lambda X :: \kappa \,.\, t)[T] \to [X \mapsto T]t \qquad (E_{\textit{t-beta}})$$

The kinding rules for the judgment $\Gamma \vdash T :: \kappa$ are likewise almost identical to the kinding rules we have seen in the specification of the kinding rules of the simply typed λ-calculus with type operators.

$$\frac{X :: \kappa \in \Gamma}{\Gamma \vdash X :: \kappa} \qquad (\kappa_{var})$$

$$\frac{\Gamma, X :: \kappa_1 \vdash T :: \kappa_2}{\Gamma \vdash \lambda X :: \kappa_1 \,.\, T :: \kappa_1 \Rightarrow \kappa_2} \qquad (\kappa_{abs})$$

$$\frac{\Gamma \vdash T_1 :: \kappa \Rightarrow \kappa' \quad \Gamma \vdash T_2 :: \kappa}{\Gamma \vdash T_1\ T_2 :: \kappa'} \qquad (\kappa_{app})$$

$$\frac{\Gamma \vdash T_1 :: * \quad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \to T_2 :: *} \qquad (\kappa_{arr})$$

$$\frac{\Gamma, X :: \kappa \vdash T :: *}{\Gamma \vdash \forall X :: \kappa \,.\, T :: *} \qquad (\kappa_{all})$$

Similarly, the rules for determining type equivalence for the relation $S \equiv T$ are largely repeated, with the exception of the rules Q_{all} and Q_{eta}, which have been added.

$$T \equiv T \qquad (Q_{\textit{refl}})$$

$$\frac{T \equiv S}{S \equiv T} \qquad (Q_{symm})$$

$$\frac{S \equiv U \quad U \equiv T}{S \equiv T} \qquad (Q_{trans})$$

$$\frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 \to S_2 \equiv T_1 \to T_2} \qquad (Q_{arr})$$

$$\frac{S \equiv T}{\forall X :: \kappa \,.\, S \equiv \forall X :: \kappa \,.\, T} \qquad (Q_{all})$$

$$\frac{S \equiv T}{\lambda X :: \kappa \,.\, S \equiv \lambda X :: \kappa \,.\, T} \qquad (Q_{abs})$$

$$\frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1\ S_2 \equiv T_1\ T_2} \qquad (Q_{app})$$

$$(\lambda X :: \kappa \,.\, T_1)\ T_2 \equiv [X \mapsto T_2]\ T_1 \qquad (Q_{beta})$$

$$\frac{X \notin T}{\lambda X :: \kappa \,.\, T\ X \equiv T} \qquad (Q_{eta})$$

Lastly, we present the typing rules for the judgment $\Gamma \vdash t : T$. The rules $T_{\textit{t-abs}}$ and $T_{\textit{t-app}}$ where modified, compared with their definition in the System F specification.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad (T_{var})$$

$$\frac{\Gamma \vdash T_1 :: * \quad \Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1 \,.\, t : T_1 \to T_2} \qquad (T_{abs})$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2} \qquad (T_{app})$$

$$\frac{\Gamma, X :: \kappa \vdash t : T}{\Gamma \vdash \lambda X :: \kappa \,.\, t : \forall X :: \kappa \,.\, T} \qquad (T_{\textit{t-abs}})$$

$$\frac{\Gamma \vdash t : \forall X :: \kappa \,.\, T_1 \quad \Gamma \vdash T_2 :: \kappa}{\Gamma \vdash t[T_2] : [X \mapsto T_2]T_1} \qquad (T_{\textit{t-app}})$$

$$\frac{\Gamma \vdash t : S \quad S \equiv T \quad \Gamma \vdash T :: *}{\Gamma \vdash t : T} \qquad (T_{\textit{t-eq}})$$

The specification of System F$_\omega$ we have just presented is based on Pierce [37]. Due to the specification of Agda, which follows in the next subsection, we are going to use an extended specification of System F$_\omega$ that is more closely aligned with the existing specification of Agda. The purpose of this extension of System F$_\omega$, which will be presented further below, is to make it easier to describe the eventual translation rules.

4.3 The specification of Agda

Only a very few programming languages have been formally specified. The most prominent example is Standard ML [32]. Another example would be Scala [35], which is not a purely functional programming language, however.[8] Like so many other programming languages, Agda is not formally specified. The following is therefore based on the existing implementation of Agda as well as personal communication with Andreas Abel.[9]

Dependently-typed programming languages do not distinguish between kinds, types, and terms. As will soon become obvious, this does not lead to a simpler grammar, as several definitions, which we did not encounter in any of the previously discussed λ-calculi, are introduced in the Agda grammar below.

$t, u, v ::=$		terms:
	x	variable
	$\lambda x \,.\, t$	abstraction
	$t\ u$	application
	$(x : U) \to T$	dependent function type
	ϕ	constant
	Set_t	universe of level t
	$Level$	type of level
	$t \sqcup u$	maximum of two levels
	$suc\ t$	successor of a level
	0	Level 0

$\phi ::=$		constants:
	c	constructor
	D	data type
	f	defined constant

[8] Having a formal specification did not protect Scala from the negative repercussions of mutable state. Given that the current Scala compiler is hardly free of bugs that are related to mutability, this means that either the formal specification of Scala is insufficient, or the formal specification of Scala is sufficient, but was not implemented properly. There is a third possibility, which is even less flattering. The interested reader may want to watch Paul Phillips's related 2013 talk "We're doing it all wrong", which is available at: https://www.youtube.com/watch?v=TS1lpKBMkgg (accessed March 30, 2015). Paul Phillips was the main contributor to the Scala language. He has since then moved on to develop his own Scala fork named *policy*, which aims to correct the fundamental issues of the current Scala implementation: https://github.com/paulp/policy.

[9] Note that we are making some simplifications in order to make the presentation clearer. For instance, we have removed record types from the specification of Agda. This does not correspond to the existing Agda implementation, but record types can be simulated with the language constructs presented below, if needed. By focusing on a subset of Agda, we are, however, able to more clearly specify translation rules. Further, by focussing on a smaller subset of Agda, a prototypical compiler to System F$_\omega$ could be developed more quickly, and then used as a foundation for future work.

Set is a signifier that is used by the current Agda implementation. The subscript of a set indicates the *level* of a kind. The purpose of the stratification of kinds into levels is to avoid impredicativity, i.e. self-referentiality, in Agda. Concretely, this means that an Agda universe, a Set, cannot be contained in itself. Instead, Set_i is contained in Set_{i+1}. If this was not the case, and universes could contain themselves, then Russel's paradox would rear its ugly head.[10]

The typing rules of Agda are:

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \qquad (T_{var})$$

$$\frac{\Gamma, x : U \vdash t : T}{\Gamma \vdash \lambda x \,.\, t : (x : U) \to T} \qquad (T_{abs})$$

$$\frac{\Gamma \vdash t : (x : U) \to T \quad \Gamma \vdash u : U}{\Gamma \vdash t\ u : [x \mapsto u]T} \qquad (T_{app})$$

$$\frac{(\phi : T) \in \Sigma}{\Gamma \vdash \phi : T} \qquad (T_{\phi})$$

$$\frac{}{\Gamma \vdash Level : Set_0} \qquad (T_{lvl})$$

$$\frac{\Gamma \vdash t : Level \quad \Gamma \vdash u : Level}{\Gamma \vdash t \sqcup u : Level} \qquad (T_{l\text{-}max})$$

$$\frac{\Gamma \vdash t : Level}{\Gamma \vdash suc\ t : Level} \qquad (T_{l\text{-}suc})$$

$$\frac{}{\Gamma \vdash 0 : Level} \qquad (T_{l\text{-}zero})$$

$$\frac{\Gamma \vdash t : Level}{\Gamma \vdash Set_t : Set_{suc\ t}} \qquad (T_{univ})$$

$$\frac{\Gamma \vdash U : Set_u \quad \Gamma, x : U \vdash T : Set_t}{\Gamma \vdash (x : U) \to T : Set_{u \sqcup t}} \qquad (T_{l\text{-}fun})$$

In the rules above, the symbol Σ represents the global signature for constants in Agda. In the rule T_ϕ, the type T of ϕ has to be looked up in Σ.

We are going to work with this specification of Agda from now on. However, System F$_\omega$, as we have previously described it, is not a suitable target for a translation, considering our specification of Agda. To remedy this situation, we are therefore going to extend System F$_\omega$ with several new language constructs.

4.4 An extended specification of System F$_\omega$

Because the previous specification of System F$_\omega$ is too sparse to serve as a viable compilation target for Agda, we are going to add several constructs to it. From now on, whenever we refer to System F$_\omega$, we refer to this extended version of System F$_\omega$ instead of the previously described specification.

With regards to the terms in the extended specification of System F$_\omega$, we introduce constructors, defined constants, as well as coercions. Types are enriched by adding data types, unit types, as well as an unknown type Any, to represent an Agda type that cannot be represented in System F$_\omega$. The addition of the unit type as well as the unit kind are both expressed as $()$. They are both necessary

[10] Russel's paradox relates to set theory. Expressed informally, it states that the set of all sets that do not contain themselves as members contains itself as a member if and only if it does not contain itself as a member.
By using the language pragma `type-in-type`, Russel's paradox can be expressed in Agda. As is to be expected, this would lead to inconsistencies.

for intermediate steps in the extraction from Agda to System F$_\omega$. Several examples much further below will provide an illustration.

$t, u ::=$		terms:
	x	variable
	$\lambda x : T \,.\, t$	abstraction
	$t\ u$	application
	$\lambda X :: \kappa \,.\, t$	type abstraction
	$t\ [T]$	type application
	ϕ	constant
	$\mathit{coerce}\ t$	coercion
$\phi ::=$		constants:
	c	constructor
	f	defined constant
$v ::=$		values:
	$\lambda x : T \,.\, t$	abstraction value
	$\lambda X :: \kappa \,.\, t$	type abstraction value
$T, U, V ::=$		types:
	X	type variable
	$T \rightarrow U$	function type
	$\forall X :: \kappa \,.\, T$	universal type
	$\lambda X :: \kappa \,.\, T$	operator abstraction
	$T\ U$	operator application
	D	data type
	f	defined type
	Any	unknown type
	$()$	unit type
$\kappa ::=$		kinds:
	$*$	kind of proper types
	$\kappa \Rightarrow \kappa$	kind of operators
	$()$	unit kind
$\Gamma ::=$		contexts:
	$\varnothing$	empty context
	$\Gamma, x : T$	context, term variable binding
	$\Gamma, X :: \kappa$	context, type variable binding

Due to the changed grammar, the typing rules need to be modified. The typing rules for the judgment $\Gamma \vdash t : T$ are given below. We omit typing rules for type equality for the sake of brevity. This affects the rule $T_{t\text{-}eq}$, which contains $S \equiv T$ as a premise.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad (T_{var})$$

$$\frac{\Gamma \vdash T :: * \quad \Gamma, x : T \vdash t : U}{\Gamma \vdash \lambda x : T \,.\, t : T \rightarrow U} \qquad (T_{abs})$$

$$\frac{\Gamma \vdash t : T \rightarrow U \quad \Gamma \vdash u : T}{\Gamma \vdash t\ u : U} \qquad (T_{app})$$

$$\frac{\Gamma, X :: \kappa \vdash t : T}{\Gamma \vdash \lambda X :: \kappa \,.\, t : \forall X :: \kappa \,.\, T} \qquad (T_{t\text{-}abs})$$

$$\frac{\Gamma \vdash t : \forall X :: \kappa \,.\, T \quad \Gamma \vdash U :: \kappa}{\Gamma \vdash t[U] : [X \mapsto U]T} \qquad (T_{t\text{-}app})$$

$$\frac{\Gamma \vdash t : S \quad \Gamma \vdash S :: *}{\Gamma \vdash \mathit{coerce}\ t : T} \qquad (T_{coer})$$

$$\frac{(\phi : T) \in \Sigma}{\Gamma \vdash \phi : T} \qquad (T_{\phi})$$

$$\frac{\Gamma \vdash t : S \quad S \equiv T \quad \Gamma \vdash T :: *}{\Gamma \vdash t : T} \qquad (T_{t\text{-}eq})$$

The symbol Σ was explained in the previous subsection on Agda; in our modified System F_ω specification, Σ likewise represents the global signature for constants.

4.5 Extraction rules for compiling Agda to System F$_\omega$

With this rather extensive amount of preliminary material behind us, we can now proceed with specifying the extraction from Agda to System F$_\omega$. The aim is to produce, given a valid Agda program as input, a valid System F$_\omega$ program that exhibits the same behaviour. In the following, $a \searrow b$ is to be read as "domain a extracts to range b".

To make the rules more easily readable, we signify Agda terms with the letters T, U, V, as well as t, u. On the other hand, System F$_\omega$ types are signified as A, B, F, G and terms as a, b.

The extraction of kinds is defined as a function, using Haskell-like syntax. The underscore is a placeholder, indicating irrelevant arguments.

$$\begin{aligned}
&\mathit{kind}(\mathit{Set}\ \_) = * \\
&\mathit{kind}((x : U) \rightarrow V) = \mathit{kind}\ U \Rightarrow \mathit{kind}\ V \\
&\mathit{kind}(\_) = ()
\end{aligned}$$

A set of any level is extracted to a proper kind, and a function type to the kind of its domain mapped to the kind of its range. Any type that is not captured by these two definitions cannot be represented as a kind in System F$_\omega$ and therefore has to be discarded. For anyone not familiar with ML-style pattern matching, we would like to add that the evaluation of the function kind proceeds from top to bottom, meaning that at first an attempt is made to match the argument with the first definition, which will succeed with any argument Set_n. Should this match fail, the function kind attempts to match by using the second definition, which will succeed for every dependent function type. If this match fails as well, then kind will attempt to match the argument with the third definition. Since the underscore character matches all inputs, this operation is bound to succeed.

The extraction from an Agda term to a System F$_\omega$ term with the judgment $\Gamma \vdash t : T \searrow F :: \kappa$ is as follows.

$$\frac{\begin{array}{c}\Gamma \vdash U : \mathit{Set}_t \searrow A :: * \\ \Gamma, x : U \vdash T : \mathit{Set}_t \searrow B :: *\end{array}}{\Gamma \vdash ((x : U) \rightarrow T) : \mathit{Set}_t \searrow (A \rightarrow B) :: *}$$

$$\frac{\mathit{kind}\ U = \kappa \neq () \quad \Gamma, x : U \vdash T : \mathit{Set}_t \searrow B : *}{\Gamma \vdash ((x : U) \rightarrow T) : \mathit{Set}_t \searrow (\forall x :: \kappa \,.\, B) :: *}$$

$$\frac{(x : T) \in \Gamma \quad \mathit{kind}\ T = \kappa \neq ()}{\Gamma \vdash x : T \searrow x :: \kappa}$$

$$\frac{(D : T) \in \Sigma \quad \mathit{kind}\ T = \kappa}{\Gamma \vdash D : T \searrow D :: \kappa}$$

$$\frac{(f : T) \in \Sigma \quad \mathit{kind}\ T = \kappa \neq ()}{\Gamma \vdash f : T \searrow f :: \kappa}$$

$$\frac{}{\Gamma \vdash \mathit{Level} : \mathit{Set_0} \searrow () :: *}$$

$$\frac{}{\Gamma \vdash \mathit{Set_t} : \mathit{Set_{suc\ t}} \searrow () :: *}$$

$$\frac{\begin{array}{l}\Gamma \vdash t : (x : U) \to T \searrow F :: \kappa \Rightarrow \kappa' \\ \Gamma \vdash u : U \searrow G : \kappa\end{array}}{\Gamma \vdash t\, u : [x \mapsto u]T \searrow F\, G :: \kappa'}\ \kappa \neq () \text{ or } F = D\vec{T}$$

$$\frac{\Gamma, x : U \vdash t : T \searrow F :: \kappa' \quad \mathit{kind}\ U = \kappa}{\Gamma \vdash \lambda x\ .\ t : (x : U) \to T \searrow (\lambda X :: \kappa\ .\ F) :: \kappa \Rightarrow \kappa'}$$

However, not all Agda terms can be represented in System F_ω. The following rules specify cases where the extraction from Agda terms to System F_ω kinds does not succeed and will have to eventually rely on either type coercion, in the case of Any, or erasure, considering that the unit kind is vacuous.

$$\frac{}{\Gamma \vdash t : \mathit{Level} \searrow () :: ()}$$

$$\frac{}{\Gamma \vdash c : T \searrow () : ()}$$

$$\frac{(x : T) \in \Gamma \quad \mathit{kind}\ T = ()}{\Gamma \vdash x : T \searrow () :: ()}$$

$$\frac{(f : T) \in \Sigma \quad \mathit{kind}\ T = ()}{\Gamma \vdash f : T \searrow () :: ()}$$

$$\frac{\Gamma \vdash t : (x : U) \to T \searrow F :: () \Rightarrow \kappa}{\Gamma \vdash t\, u : [x \mapsto u]T \searrow \mathsf{Any} :: \kappa}\ F \neq D\vec{T}$$

$$\frac{\Gamma \vdash t : (x : U) \to T \searrow () :: ()}{\Gamma \vdash t\, u : [x \mapsto u]T \searrow () :: ()}$$

$$\frac{\Gamma, x : U \vdash t : T \searrow () :: ()}{\Gamma \vdash \lambda x\ .\ t : (x : U) \to T \searrow () :: ()}$$

In addition to extraction rules for kinds we also need rules for the extraction from an Agda term to a function type in System F_ω. The corresponding judgment is $\Gamma \vdash t : T \searrow a : A$ where $\Gamma \vdash T : \mathit{Set} \searrow A :: *$.

$$\frac{\Gamma \vdash U : \mathit{Set_0} \searrow A : *}{\Gamma \vdash t : (x : U) \to V \searrow \lambda(x : A)\ .\ b : A \to B}$$

$$\frac{\mathit{kind}\ U = \kappa \neq () \quad \Gamma, x : U \vdash t\, x : V \searrow b : B}{\Gamma \vdash t : (x : U) \to V \searrow (\lambda x :: \kappa\ .\ b) : (\forall x :: \kappa\ .\ B)}$$

$$\frac{(x : T) \in \Gamma \quad \Gamma \vdash T : \mathit{Set_0} \searrow A :: *}{\Gamma \vdash x : T \searrow x : A}$$

$$\frac{(x : T) \in \Gamma \quad \Gamma \vdash T : \mathit{Set_0} \searrow () :: \_}{\Gamma \vdash x : T \searrow () : \mathsf{Any}}$$

$$\frac{\begin{array}{l}\Gamma \vdash t : (x : U) \to T \searrow b : A \to B \\ \Gamma \vdash u : U \searrow a : A\end{array}}{\Gamma \vdash t\, u : [x \mapsto u]T \searrow b\, a : B}$$

$$\frac{\begin{array}{l}\Gamma \vdash t : (x : U) \to T \searrow b : C \\ \Gamma \vdash u : U \searrow a : A\end{array}}{\Gamma \vdash t\, u : [x \mapsto u]T \searrow (\mathit{coerce}\ b)\ a : \mathsf{Any}}\ C \neq A \to B$$

$$\frac{(c : T) \in \Sigma \quad \Gamma \vdash T : \mathit{Set_0} \searrow A :: *}{\Gamma \vdash c : T \searrow c : A}$$

$$\frac{}{\Gamma \vdash \mathit{Set_t} : \mathit{Set_{suc\ t}} \searrow () : \mathsf{Any}}$$

4.6 Compilation examples

After specifying extraction rules from Agda to System F_ω, we can now proceed with the presentation of several examples that illustrate the intended results of a compiler that follows those rules. Several caveats are in order, however. In order to present slightly more interesting examples, we have to occasionally postulate the existence of various data types as well as language constructs that have not been covered in the previous dicussion. This hardly in the spirit of Agda, considering that many tutorials start *ab ovo* and do not even assume natural numbers or booleans as given [7, 34]. In the following examples, the starting point is valid Agda source code, which will then be juxtaposed with its translation into System F_ω.

As a warmup, we will start with the definition of a control structure for if-then-else statements. Agda allows *mixfix* operators [16], which are illustrated in the first example. However, our starting point will be a more conservative definition of the same control statement that can be directly translated into System F_ω. Of course we are also assuming that we have already defined boolean data types.[11]

Using mixfix-operators, an idiomatic Agda definition of the if-then-else statement would be as follows.

```
if_then_else_ : {A : Set} -> Bool -> A -> A -> A
if true  then x else y = x
if false then x else y = y
```

However, we can also use a definition that looks a lot more familiar to a programmer with a background in common functional programming languages. Note that the following code listing assumes the existence of pattern matching. The function body represents a λ-expression. While Agda allows Unicode characters in source code, LaTeX unfortunately does not. Expressed in the syntax we have been using so far, the first function definition would be written as $\lambda s_1\ s_2\ .\ s_1$. The sometimes great similarity between Agda and resulting System F_ω code may lead to some confusion, which is why we will use an arrow in λ-expressions in Agda, but a dot in the corresponding expressions in System F_ω.

```
if-then-else : {A : Set} -> Bool -> A -> A -> A
if-then-else true  = \ s1 s2 -> s1
if-then-else false = \ s1 s2 -> s2
```

The translation to System F_ω is relatively straightforward. The universe, here implied to be $\mathit{Set_0}$ is extracted to the proper kind $*$ that is quantified over all instances of the type variable A. It would

[11] For a programmer coming from an imperative background, it may come as a surprise that data types that are provided as primitives in their language of choice, like booleans or integers, can be defined in a language like Haskell or Agda.

certainly be pleasant to have an implementation of System F_ω with pattern matching, and if we had one, the following code would be the result.

```
if-then-else : forall A :: * . Bool -> A -> A -> A
if-then-else true  = \  s1 s2 . s1
if-then-else false = \  s1 s2 . s2
```

In its purest form, though, type polymorphism is arguably best represented by the identity function. This furthermore has the side effect of ensuring some continuity with the earlier parts of this paper. In Agda, the identity function can be represented as follows.

```
id : (A : Set) -> A -> A
id = \ A -> \ x -> x
```

Again, in System F_ω the type universe is represented as a proper kind.

```
id : forall A :: * . A -> A
id = \ A :: * . \ x : A . x
```

Functions that do not rely on type polymorphism are virtually identical in Agda and System F_ω. To build upon the previous work, while simultaneously undoing decades of progress in programming language theory, we could define an if-then-else construct that only works for natural numbers, like the following example. The function definition in Agda is followed by a possible representation in System F_ω.

```
if-then-else : Bool -> Nat -> Nat -> Nat
if-then-else true  = \ s1 s2 -> s1
if-then-else false = \ s1 s2 -> s2


if-then-else : Bool -> Nat -> Nat -> Nat
if-then-else true  = \  s1 s2 . s1
if-then-else false = \  s1 s2 . s2
```

Of course, more interesting monomorphic functions can be defined as well, like the factorial function, which assumes that we have defined natural numbers as well as a multiplication function. The code would look the same in Agda and System F_ω, apart from possible cosmetic differences.

```
fact : Nat -> Nat
fact zero     = succ zero
fact (succ n) = mult (succ n) (fact n)
```

We spent a considerable amount of time on singleton lists. The following code example first gives a data type definition of lists, and afterwards the definition of a function that creates a singleton list when given any value as input. Since we are now dealing with a more complex structure, we will take a glance at a possible internal representation in Agda as well, right after showing the typical source code representation.

```
data List (A : Set) : Set where
  nil  : List A
  cons : A -> List A -> List A

List : (A : Set_0) -> Set_0
nil  : (A : Set_0) -> List A
cons : (A : Set_0) (x : A) (xs : List A) -> List A
```

In System F_ω, the list data type would have the following representation, which again shows the extraction from Agda types to System F_ω kinds:

```
List : forall A :: * -> *
nil  : forall A :: * -> List A
cons : forall A :: * . A -> List A -> List A
```

Much earlier in this paper we have encountered a function for creating singleton lists. This example is repeated below, first in Agda, and then followed by a corresponding extraction to System F_ω. Note that, in the System F_ω code, type application is indicated by square brackets.

```
makeSingleton : (A : Set_0) (x : A) -> List A
makeSingleton = \ A \ x -> cons A x (nil A)

makeSingleton : forall A :: * . A -> List A
makeSingleton = \ A :: * .
                \ x : A . cons [A] x (nil [A])
```

A relatively large part of the extraction rules is concerned with Agda types that cannot be represented in System F_ω. The next example will illustrate this in more practical terms. We have already seen that universes, written as Set_t in Agda, are represented as proper kinds in System F_ω. Now we will also encounter the case of a type that cannot be expressed in System F_ω and therefore has to be represented as Any instead.

```
case : (l : Level) (T : Nat -> Set_l) (n : Nat)
           (z : T 0) (s : (x : Nat) (y : T x)
               -> T (suc x)) -> T n


case : () -> forall ( T : ( ) -> *) . Nat -> Any ->
          (Nat -> Any -> Any) -> Any
```

The next example combines several of the necessary translations we have discussed so far, and adds a new data type as well. This code, given in its internal representation in Agda, computes the sum of two natural numbers through recursion.

```
rec : (l : Level) (T : Nat -> Set l) (z : T 0)
        (s : (m : Nat) -> T m -> T (suc m))
        ->  (n : Nat) -> T n
rec l [T] z s 0       = z
rec l [T] z s (suc n) = s n (rec l [T] z s n)

Sum : Nat -> Set
Sum = rec 0 (\ _ -> Set) Nat (\ _ R -> Nat -> R)

sum : (n : Nat) (acc : Nat) -> Sum n
sum = rec 0 (\ n -> Nat -> Sum n) (\ acc -> acc)
                (\ _ r acc m -> r (acc + m))
```

Please note that we, again, assume pattern matching facilities in System F_ω in the definition of `rec`.

```
rec : () -> forall (T : () -> *) -> Any ->
        (Nat -> Any -> Any) -> Nat -> Any
rec l T z s 0       = z
rec l T z s (suc n) = s n (rec l T z s n)

Sum : () -> *
Sum = Any

sum : Nat -> Nat -> Any
sum = rec 0 () (\ acc -> coerce acc)
          (\ _ r acc m -> (coerce r) (acc + coerce m))
```

As the last example we would like to present the lookup function for a polymorphic length-indexed vector. This presumes that we have a definition for vectors, natural numbers, and finite sets of natural numbers.

```
lookup : {A : Set} {n : Nat} -> Vec A n -> Fin n -> A
lookup (x :: xs) fzero   = x
lookup (x :: xs) fsucc i = lookup xs i


lookup :  forall A :: * .
          Nat -> Vec A () -> Fin () -> A
lookup [A] (succ n) (x :: xs) fzero   = x
lookup [A] (succ n) (x :: xs) fsucc i =
  lookup [A] n xs i
```

The previous example may not look overly interesting at first, but it has potentially far-reaching implications, which we will discuss in the final section of this paper.

5. Discussion and outlook

In this paper we have proposed extraction rules that form the theoretical foundations for a future compiler backend for Agda that performs *type-directed* translation to System F_ω. This means that we will retain as much as possible of the type annotations that are present in Agda source code, and only resort to the catch-all type Any when strictly necessary. System F_ω is only an intermediary step, though, as the desired goal of a new compiler backend for Agda would be to generate Haskell source code.

The benefits of this approach are two-fold. First, the generated Haskell code should be quite readable. In fact, the goal would be to produce Haskell code that does not look too dissimilar to what a human programmer would produce. By comparison, the output of the current Agda backend that targets Haskell, MAlonzo, uses the Haskell primitive `unsafeCoerce` for every type. On the other hand, the code generated by a type-directed compiler can be expected to be much more readable, which would lead to a more productive feedback loop, as we would be in a position to use the generated output to more easily assess the quality of the compiler. The second clear benefit of our approach is that the Haskell compiler GHC would be able to perform optimisations more or less unhindered, as it could perform optimisation passes that rely on type information.

As a practical consequence, the desired compiler could be used to generate relatively efficient Haskell code. One side effect may be that more users get interested in working with Agda, if the output of a new future backend was better, i.e. more readable, and led to code that executes faster, due to GHC being able to perform various optimisations that rely on type information. From the perspective of a software engineer who is concerned about security and correctness guarantees, there is another rather significant benefit as well: Agda could be used to produce Haskell code that meets stronger correctness guarantees than Haskell itself is able to provide, since any program that gets compiled to Haskell would first have to satisfy the Agda type checker and termination checker. Due to the lack of Turing completeness in Agda, this means that fewer programs could be written than in plain Haskell. On the other hand, those programs would provide stronger guarantees.

At the end of the previous section we have seen an example of a lookup function on length-indexed vectors. Concretely, this means that one could write such a function in Agda, rely on Agda's type system and termination checker to ensure the correctness of the function definition, and then compile this code to System F_ω or Haskell, respectively. In plain Haskell it is not possible to achieve these effects. Thus, by relying on Agda as a pre-processor, greater guarantees for Haskell source code could be made. Note that the often-used example of safe lookup of length-indexed vectors is hardly the culmination of what would be possible. As was mentioned before, but which would by far have exceeded the scope of this paper to discuss in detail, Agda makes it possible to encode arbitrary specifications in the type signature. More or less any property one could think of, as long as the programmer can provide a proof for it, can be devised. Further, termination checking is arguably a desirable property in many domains, but probably not when your goal is, for instance, to keep a server loop running. Thus, an Agda compiler that emits Haskell code may be an enticing proposition for some programmers.

Yet, we should not get too excited at this point. We have only covered a subset of Agda. Furthermore, in our translation examples we made various assumptions in order to facilitate a more digestible presentation of the effect of the translation rules. Assuming the existence of helpful language constructs is certainly much easier than implementing said features in an actual compiler, even one that is merely intended as a proof of concept. The next steps towards a functional System F_ω backend for Agda are indeed daunting. A possible continuation would be to implement a compiler for the subset of Agda that is discussed in this paper, with the goal of targeting an implementation of System F_ω that contains the extensions we have assumed, such as pattern matching. Once this point is reached, it would be possible to attempt a translation to Haskell source code. Should this succeed, then a second iteration could be attempted in order to add further features of Agda, such as record types, which we have omitted in our discussion. Of course this process would have to be repeated many more times. Agda is likely to gain new features that will eventually need to be supported by a hypothetical System F_ω compiler backend. Thus, Agda is a moving target, which entails that work on a new compiler backend may never be complete. Fortunately, Haskell has been standardised, with the most recent version being Haskell 2010 [26]. This means that even though a new Haskell standard may eventually appear, and the stream of extensions to the main compiler GHC may never end,[12] it would be possible to concentrate on a rather static target for the foreseeable future.

Developing a compiler that translates from a programming language with a more expressive type system into one with a less expressive one is not a new idea. The benefits of this approach, such as reducing programmer errors, if not eliminating entire classes of errors altogether, seem rather enticing. This may explain why this idea has gained significant traction in recent years. Letouzey's work [24, 25] on extracting Coq into certified programs in OCaml, Haskell, and Scheme is still the high-water mark in this area. However, there have been significant developments even in pedestrian programming languages. This might partly be a counter reaction to the infatuation with untyped programming languages, which certain corners of the software development industry have been nurturing. JavaScript is an example of a highly successful programming language, considering its wide adoption, but it is also an example, if not a paragon, of a highly unsafe programming language. The programming language C has a similar reputation.

Therefore, it may not come as a surprise that efforts have been made to make programming in inherently unsafe programming languages safer, but not by writing more unit tests and hoping that there are no dormant critical bugs left in the source code. Instead, a more promising approach seems to consist of having a compiler generate C or JavaScript code. Two relevant recent examples for source-to-source compilers are Haste [18], a Haskell compiler backend that emits JavaScript code, and Feldspar [2], an embedded domain-specific language for digital signal processing that is written in Haskell, which emits C code. In both cases, the programmer is able to make use of the type system Haskell provides, and the greater means for abstraction, which both potentially lead to greater efficiency. Likewise, it is imaginable that one day Haskell programmers may write part of their code in Agda, ensure the correctness of critical functions, and then compile their code to Haskell, with the goal of getting stronger guarantees of correctness. The resulting Haskell code would be safer because Agda is able to verify certain properties automatically. For instance, if you wanted to ensure that your Haskell program terminates, but do not have the inclination to write a termination checker, then Agda could be used for termination checking, with the effect that the Haskell code this Agda program compiles into has been verified to terminate, even though Haskell itself is unable to provide this guarantee.

Acknowledgments

First and foremost, I would like to warmly thank Andreas Abel for the extensive support and guidance he provided as the supervisor of this thesis. It was an honour and a pleasure to work under someone so knowledgeable and dedicated. Looking back, I have to admit that I was rather naïve when I first approached him, in the middle of 2014, to inquire about possible thesis topics that are related to his research. Unlike Socrates I did not even know that I knew nothing.

[12] At the time of writing, GHC supports over 80 language extensions.

Over the course of about a year, though, I gained a much deeper understanding of many areas of programming language theory, and a solid foundation for my further studies.

While I have had much less personal contact with Aarne Ranta and David Sands than with Andreas Abel, they still greatly influenced me, as their courses at Chalmers allowed me to lay the foundation for this paper. In that regard, I would like to thank Aarne Ranta for his inspired delivery of the course Programming Language Technology in spring 2014. Producing a working compiler that targets the JVM was one of the highlights of my education, not just because the underlying theory was not entirely unrelated to my background in theoretical philosophy and linguistics. It also made me realise that not everything about Java is bad. Further, the importance of David Sands' course Functional Programming for my development as a programmer cannot be overstated. In early October 2013 I still considered Scheme the pinnacle of elegance in programming language design, and was skeptical of the usefulness of static typing. By December 2013 I wondered how I was ever able to get anything done in any non-pure non-functional non-statically typed programming language.

Lastly, I would like to thank a number of friends and fellow students for the comments and feedback they provided. Daniel Lee deserves a special mention, not only for some very insightful comments on this paper, but also for nudging me towards computer science by recommending the *wizard book* to me, and his subsequent mentorship throughout the last four or five years. In addition, I received helpful feedback from, in alphabetical order, James Eagle, Niklas Logren, Malcolm Matalka, and Behrouz Talebi.

References

[1] A. Abel. foetus–termination checker for simple functional programs. *Programming Lab Report*, 474, 1998.

[2] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda. Feldspar: A domain specific language for digital signal processing algorithms. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 169–178. IEEE, 2010.

[3] H. Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991.

[4] H. P. Barendregt. *The lambda calculus*, volume 3. North-Holland Amsterdam, 1984.

[5] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, et al. The coq proof assistant reference manual–version 7.2. Technical report, Technical Report 0255, INRIA, 2002.

[6] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[7] A. Bove and P. Dybjer. Dependent types at work. In *Language engineering and rigorous software development*, pages 57–99. Springer, 2009.

[8] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(05):552–593, 2013.

[9] A. Chlipala. *Certified programming with dependent types*. MIT Press, 2011.

[10] A. Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, pages 345–363, 1936.

[11] A. Church and J. B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936.

[12] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[13] T. Coquand and G. Huet. The calculus of constructions. *Information and computation*, 76(2):95–120, 1988.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[15] K. Crary. A syntactic account of singleton types via hereditary substitution. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, pages 21–29. ACM, 2009.

[16] N. A. Danielsson and U. Norell. Parsing mixfix operators. In *Implementation and Application of Functional Languages*, pages 80–99. Springer, 2011.

[17] G. Dowek, G. Huet, and B. Werner. On the definition of the eta-long normal form in type systems of the cube. In *Informal Proceedings of the Workshop on Types for Proofs and Programs, Nijmegen, The Netherlands*, 1993.

[18] A. Ekblad. Towards a declarative web. *Master of Science Thesis, University of Gothenburg*, 2012.

[19] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to design programs*. MIT Press, 2001.

[20] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[21] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, pages 29–60, 1969.

[22] W. A. Howard. The formulae-as-types notion of construction. In J. P. Hindley, J. Roger; Seldin, editor, *To HB Curry: essays on combinatory logic, lambda calculus, and formalism*, pages 479–490. Academic Press, 1980.

[23] B. W. Kernighan and D. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988.

[24] P. Letouzey. A new extraction for coq. In *Types for proofs and programs*, pages 200–219. Springer, 2003.

[25] P. Letouzey. Extraction in coq: An overview. In *Logic and Theory of Algorithms*, pages 359–369. Springer, 2008.

[26] S. Marlow et al. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)*, 2010.

[27] Y. Matsumoto and K. Ishituka. *Ruby programming language*. Addison Wesley, 2002.

[28] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4): 184–195, 1960.

[29] S. McConnell. *Code complete*. Microsoft Press, 2004.

[30] R. Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, 4(1):1–22, 1977.

[31] R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[32] R. Milner. *The definition of standard ML: revised*. MIT Press, 1997.

[33] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[34] U. Norell and J. Chapman. Dependently typed programming in agda, 2013.

[35] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. The scala language specification, 2004.

[36] R. Patton. *Software testing*. Sams Pub., 2006.

[37] B. C. Pierce. *Types and programming languages*. MIT Press, 2002.

[38] G. D. Plotkin. Lcf considered as a programming language. *Theoretical computer science*, 5(3):223–255, 1977.

[39] J. C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, pages 408–425. Springer, 1974.

[40] P. Runeson. A survey of unit testing practices. *Software, IEEE*, 23(4): 22–29, 2006.

[41] F. Shull, V. Basili, B. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz. What we have learned about fighting defects. In *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, pages 249–258. IEEE, 2002.

[42] M. Sulzmann, M. M. Chakravarty, S. P. Jones, and K. Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66. ACM, 2007.

[43] S. Thompson. *Type theory and functional programming*. Addison Wesley, 1991.

[44] G. Van Rossum and F. L. Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[45] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework i: Judgments and properties. Technical report, DTIC Document, 2003.