ANDREJ TOKARČÍK

AN EXECUTABLE FORMAL SEMANTICS OF AGDA

AN EXECUTABLE FORMAL SEMANTICS OF AGDA

ANDREJ TOKARČÍK

Master's Thesis

Faculty of Informatics
Masaryk University

Brno, May 2015

In honour of God our Father, the almighty God,
Creator of all Things, the Most Sublime and Best Father.

Dedicated to Him with a contrite and humble heart.

DECLARATION

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

.....
Bc. Andrej Tokarčík

SUPERVISOR:
Mgr. Jan Obdržálek, Ph.D.

ACKNOWLEDGMENTS

I would like to express my gratitude to my supervisor Mgr. Jan Obdržálek, Ph.D., for his support and advice. My earnest thanks also go to my friends and family for their help and care during the period of writing the thesis.

ABSTRACT

Agda is an actively developed dependently typed programming language and interactive theorem prover based on a variant of Martin-Löf type theory. This thesis presents an executable formal semantics of a portion of the language, created using the $\mathbb{K}$ semantic framework. Issues pertaining to formalisation of Agda as well as the key decisions made during the development of the semantics are discussed.

The core of the project is an implementation of the typechecking and type inference algorithms together with support for declarations of inductively defined families of datatypes and dependently typed functions.

In addition, the covered features also include insertion of metavariables in place of implicit arguments and simplified pattern matching. The work thus demonstrates the ability to provide operational semantics of dependently typed programming languages without disregarding aspects that make their use practical.

KEYWORDS

$\mathbb{K}$, Agda, $\mathbb{K}$Agda, formal semantics, operational semantics, rewriting logic, type theory, dependent type systems

ZHRNUTIE

Agda je aktívne vyvíjaný programovací jazyk so závislými typmi a interaktívny dokazovač matematických viet založený na upravenej teórii typov Martin-Löfa. V práci je predstavená spustiteľná formálna sémantika časti tohto jazyka vytvorená pomocou sémantického frameworku $\mathbb{K}$. Diskutujú sa jednak otázky týkajúce sa formalizácie Agdy a druhak kľúčové rozhodnutia učinené počas vývoja jej sémantiky.

Základom projektu je implementácia algoritmov na kontrolu a odvodzovanie typov spolu s podporou deklarovania induktívne definovaných rodín dátových typov a závislo typovaných funkcií.

Okrem toho je zahrnuté aj vkladanie metapremenných namiesto implicitných argumentov a zjednodušený pattern matching. Práca tak ukazuje, že je možné poskytnúť operačnú sémantiku závislo typovaných programovacích jazykov aj bez zanedbania vlastností, ktoré ich robia praktickými na použitie.

KĽÚČOVÉ SLOVÁ

$\mathbb{K}$, Agda, $\mathbb{K}$Agda, formálna sémantika, operačná sémantika, prepisovacia logika, teória typov, typové systémy so závislými typmi

CONTENTS

LIST OF FIGURES

TERMINOLOGY

Agda	the dependently programming language or specifically the official type checker, depending on the context
𝕀KAgda	𝕀K semantics of Agda, presented in this thesis
GADT	generalised algebraic data type
AST	abstract syntax tree

1 INTRODUCTION

Type theories emerged as an alternative to naïve set theory after it was shown to be inconsistent. Perhaps the most notable definition that exhibits this inconsistency is that of Russell's paradox which concerns a set of all sets that are not members of themselves, $R = \{X \mid X \notin X\}$. Clearly, R is a member of itself precisely when it is not a member of itself, which is contradictory. After the possibility of such a contradictory formulation within naïve set theory was discovered, logicians and mathematicians had to reconsider their basic principles and to find ways in which set definitions are appropriately restricted.

Type theories, that is, formal approaches to the study of type systems, are the fruits of one of the directions taken to address this issue. The notion of type can be understood similarly to the notion of set except that types belong to a hierarchy of universes that avoids self-reference: the type of a universe is always of the higher level, it cannot be a member of itself.

In the meantime, type systems have been also explored and employed by designers of programming languages. The purpose of types in this field has been to indicate the intended behaviour of a program along with the implementation of that behaviour and to aid compilers, interpreters and such when it comes to finding errors and various optimisations prior to the actual execution.

Not all type systems meet those ends equally, though. To take a stock example, widely used type systems of modern languages allow one to state that a sorting function is unary with a list of numbers as its sole input and that it produces a list of numbers. The statement certainly does not reveal much about the real reason for why there is a special category of sorting functions. Indeed, even the identity function would fit such a vague description!

A driving force behind the interest in type systems in computer science is grounded in this issue of expressivity. How to state the type of a sorting function more accurately? Dependent type systems such as the one discussed in this thesis attempt to give a solution to this problem by extending the vocabulary spoken in the world of types. Types then become strikingly similar to logical formulæ, giving way to a synthesis of the endeavours of logicians and computer scientists.

Refined types mean more automated help when constructing programs and a higher degree of certainty about their correctness in the end – but they as well demand more for the machine to be convinced that the algorithms are in line with the given specification. Sophisticated type systems thus might delay the time when a program gets

finally executed but once such a state is reached there is a proportionally greater guarantee that the program works as it should. And that is a most welcome exchange in many cases.

1.1 CONTRIBUTIONS

This thesis presents a formal operational semantics of the dependently typed functional programming language and interactive theorem prover Agda. The semantics is written using the $\mathbb{K}$ semantic framework. The importance of the rigorous mathematical study of the meaning of programming languages is generally underestimated because of the perceived lack of its usefulness. $\mathbb{K}$ is a powerful formalism that generates an interpreter and other analysis tools from language definitions and so gives a practical incentive for a semantics-based approach towards the design and implementation of programming languages.

The main contributions are:

A. a discussion of issues related to formalisation of Agda,

B. the first formal semantics of a portion of Agda in its capacity as a practical programming language, and

C. the first $\mathbb{K}$ semantics of a language featuring a dependent type system.

The latter two points culminate in a single semantics project named $\mathbb{K}$Agda whose development and functionality is the subject of the thesis.

Even though the semantics is most certainly incomplete with respect to the whole of Agda's features, non-trivial programs that rely on the power of dependent type systems can be executed. $\mathbb{K}$Agda can do basic theorem proving, too.

Some features supported by $\mathbb{K}$Agda include:

- typechecking of expressions, including **let** bindings;
- declarations of (parametrised) datatypes, inductively defined families and (dependent) functions;
- simplified inference of metavariables and implicit arguments;
- simplified pattern matching over inductively defined families of datatypes.

The source code of the semantics, the test suite and related files can be found in the archive enclosed with this thesis or in their most up-to-date form online at:

`https://github.com/andrejtokarcik/agda-semantics`

To get more acquainted with the final product and its technical specifics see the quickstart guide in Appendix A.

1.2 OVERVIEW OF THE THESIS

The thesis presupposes elementary knowledge of principles of programming language design and functional programming. A general familiarity with theoretical computer science should be a sufficient background.

Chapter 2 discusses motivation for using $\mathbb{K}$ as the semantic framework. Its philosophy, notation, pros and cons are briefly discussed. The reader will become acquainted with the structure of $\mathbb{K}$ language definitions, basic rewriting rules and methods of argument evaluation.

Chapter 3 is an introduction to features of the Agda language. Dependent types and their characteristic attributes are explained and demonstrated on examples. The relationship between dependent type systems and constructive logics and the consequences on machine-checked theorem proving are also examined.

The previous two chapters are to be understood as preliminary expositions of the technologies that are integrated together within $\mathbb{K}$Agda. Their purpose is to make the reader capable to follow the rest of the text that builds upon this basic knowledge. Since the explanations are by no means exhaustive or self-contained, they are supplemented by references to other literature.

The core of the thesis is Chapter 4. After the challenges of defining Agda's semantics are stated, I continue with how the syntactic and semantic parts of the semantics are approached, where the latter part is primary and much longer. The chapter finishes with a section on how a dependent type theory is realised in $\mathbb{K}$Agda. The correspondence between the $\mathbb{K}$ rewriting rules and the type-theoretic rules is shown on the examples of type inference for Π-abstractions and dependent function applications.

Chapter 5 concludes the whole thesis and analyses the achievements and limitations of the semantic definition.

At last, Appendix A is a technical guide on how to set up the $\mathbb{K}$Agda system and how to perform typechecking of Agda programs with it.

2

𝕂 FRAMEWORK

In an ideal world, every programming language has a formal semantic definition that is simple, concise, complete and modular. Furthermore, a variety of tools would be automatically derived from it: parsers, interpreters, compilers, debuggers, state-space explorers, model checkers, and so forth. Not only would it be possible to reason formally about the properties of programs and programming languages but lots of duplicated effort would be prevented as all the language-specific software would be replaced by a generic semantics-based toolset once and for all.

𝕂 [31] is a framework for defining operational semantics of programming languages that aims to bring this ideal world closer. Its implementation [32], using which the semantics discussed in this thesis has been created, comprises a collection of software tools for translating language definitions to rewriting-logic theories and for their subsequent execution or analysis [18]. Several languages have been formalised in 𝕂, including (in the order of publication) C [14], Python [17], PHP [15], Java [9], and most recently JavaScript [28].

The main assets of 𝕂 are its convenient and modular notation for defining both the syntax and the semantics of a programming language on the one hand and its generic tools that work with these definitions on the other hand. Even though the scope of the 𝕂-provided tools is broader, the following have been used in connection with the development of 𝕂Agda:

- `kompile` that compiles programming language definitions into Maude modules;
- `krun` that works as a parser and interpreter of programs written in the `kompiled` programming language;
- `ktest` that tests the language definition by `krunning` a set of programs and comparing their results with expected outcomes.

For more details on their usage refer to Appendix A.2 at page 48.

Perhaps the greatest disadvantage has to do with performance issues. It is indeed true that 𝕂's generic tools are given 'for free' once the semantics is created but it does not follow that these would be *ipso facto* comparable or even preferable to the tailored compilers, typecheckers, etc. Performance loss is not always negligible and must be taken into account.

This chapter explains the basics of the 𝕂 formalism and notation that are necessary to understand the portions of 𝕂Agda discussed in the rest of the thesis. For a more thorough exposition of the framework, see its overview [31] and take its tutorial [30].

2.1 LANGUAGE DEFINITIONS

A language definition consists of a syntax and a semantics of the language. In $\mathbb{K}$, a syntax is comprised of syntactic *sorts* given using BNF annotated with semantics attributes. A semantics is given as a set of rewriting *rules* over a *configuration*. A configuration is an algebraic structure holding all information about the program state, organised as a (possibly nested) set of *cells* (fast-forward to Figure 4 at page 27 for an example).

Cells are containers of data on which the semantics operates.

The `k` cell is especially important in this regard. It encloses a sequence of computations to be performed, usually pieces of the semi-processed program text, with the top (or leftmost) item determining the current computation phase. Items of the sequence are delimited by '$\curvearrowright$' with the unit $\bullet_K$. The unit matches on any position in the sequence: for instance, the sequence $K \curvearrowright \ldots$ is matched by the pattern that expects K to be at the top of the sequence but also by the pattern that expects $\bullet_K \curvearrowright K$ to be at the top.[1]

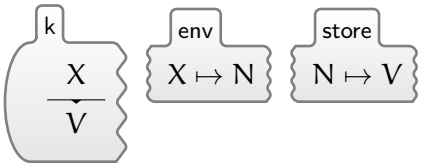
Since the `k` cell is initialised with the text of an input program, the rules generate an input-dependent transition system. A single path in the system characterises one possible way to interpret the particular program.

2.2 REWRITING RULES

Rewriting rules contain patterns to match the contents of configuration cells. A rule that matches can be triggered with the effect of the configuration's being updated as specified by the rule.

We consider the following rule:

RULE VAR-LOOKUP



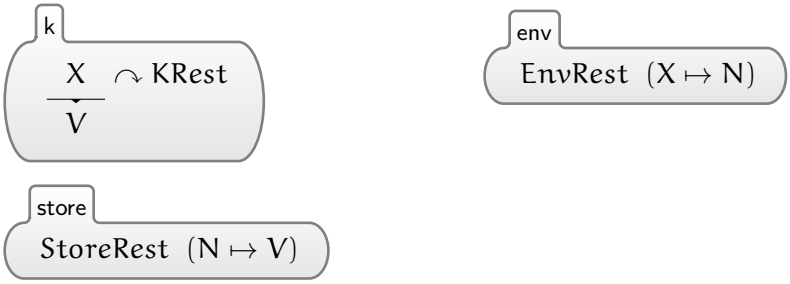
The rule is immediately observed to involve three configuration cells. If there are any other cells in the configuration, they are assumed to be irrelevant and match automatically.

What exactly is being matched here, though? To clear up possible confusion in advance, the '$\mapsto$' symbol does not refer to the rewriting relation. Instead, the relation is shown as the horizontal line below X, whereas '$\mapsto$' is used to denote items of a map: the `env` and `store` cells therefore both store a map. A cell with no horizontal line is only read but not rewritten by the rule, which is again the case of `env` and `store`. Variables start with a capital letter.

1 The latter pattern is useful for putting new items in front of the current `k` top as in the 2nd rule from Figure 10 at page 36.

Also notice the 'torn' sides of the cells conveying that the cell can contain data besides those items listed. The left side of the k cell is not torn, however, and so the left boundary must correspond with the left end of the data structure: for the sequence in k it means that X must be precisely its leftmost (or top) item. On the other hand, since env and store are ruptured from both sides, the matched map items can be located wherever in the map. If we wanted to be redundantly explicit we could rewrite the rule with additional variables thus:

RULE VAR-LOOKUP-EXPLICIT



All in all, both of the rules say that if the top of k is some thing X, which is a key for another thing N in the env map, which in turn is also a key for yet another thing V in store, then X in k should be rewritten into V. In more human terms, the rule performs a two-level lookup. If we took the rule's name seriously, we could deduce that the programming language that is being described contains variables and the things looked up and stored are in fact these variables. The env cell could be then understood as mapping variables to addresses while store would be mapping addresses to actual values. *Voilà*, we really seem to have just defined the semantics of a variable lookup!

What if X and V turned out to be equal?

The example has already touched upon an important aspect of how IK promotes modularity: that not all the configuration cells are required to be mentioned in rules and even then only the relevant portions of those mentioned have to be explicitly included.

2.3 EVALUATION CONTEXTS

Now we examine the two rules for the conditional expression:

RULE IF-TRUE

$$\frac{\texttt{if}\ B\ \texttt{then}\ E\ \texttt{else}\ \_}{E}$$

requires $B =_K \texttt{true}$

RULE IF-FALSE

$$\frac{\texttt{if}\ B\ \texttt{then}\ \_\ \texttt{else}\ E}{E}$$

requires $B =_K \texttt{false}$

The rules do not reference any cell at all, how come? Such rules are understood simply to match on the k cell by default. The symbol '—' represents a wildcard pattern that always matches and about whose value we don't care.

Side conditions are also attached to the rules here. Each of the rules can get triggered only if its side condition is met, i. e., it evaluates as

`true`. The conditions here are quite elementary, though, so one might wonder why bother and not use pattern matching directly. Indeed, we could equivalently say:

RULE IF-TRUE-MATCHING

$$\frac{\texttt{if true then E else} \_}{\texttt{E}}$$

RULE IF-FALSE-MATCHING

$$\frac{\texttt{if false then} \_ \texttt{ else E}}{\texttt{E}}$$

Remarkably, the rules seem to presuppose their condition expression to be booleans in their canonical forms[2] only, contrary to the intuition that `if`s with a condition like $2 \leqslant 3$ should certainly evaluate to the first branch as if the condition were directly `true`. The rules seem to accept nothing else but the two trivial cases.

That argument would hold if the rules were intended to be self-sufficient. Far from it: they are usually coupled with another set of rules that takes care of evaluation of the arguments into their 'final' form. In other words, there are supplementary rules that secure that a language construct is *strict* in its arguments, i. e., the arguments are evaluated before dealing with the construct itself.

For the `if` construct, the correct evaluation strategy is guaranteed by the pair of rules:

RULE IF-HEAT

$$\frac{\bullet_K}{\texttt{C}} \curvearrowright \texttt{if } \frac{\texttt{C}}{\square} \texttt{ then} \_ \texttt{ else} \_$$

requires $\texttt{isBool}(\texttt{C}) \neq_K \texttt{true}$

RULE IF-COOL

$$\frac{\texttt{C}}{\bullet_K} \curvearrowright \texttt{if } \frac{\square}{\texttt{C}} \texttt{ then} \_ \texttt{ else} \_$$

requires $\texttt{isBool}(\texttt{C}) =_K \texttt{true}$

The first rule is a 'heating' rule that is executed only if the condition C is not a final result (here a canonical boolean). In such a case, C gets lifted on top of the `k` computation stack with its position in the original expression marked by a 'hole' ($\square$).

Being on top, C is matched by other rewriting rules that (hopefully) reduce it into a canonical boolean value. When the transformations finish, the second, 'cooling' rule is triggered, which brings the produced boolean back to its original context. Afterwards, the actual rules for the `if` construct from the beginning of the section are guaranteed to match.

Since the heating/cooling rules are very commonly required yet can be very tedious to write, $\mathbb{K}$ aids the semanticist by providing him with more concise means to specify evaluation requirements of a

2 Elements of sorts or types are generally said to be in canonical form when they are built solely by constructors, whereas non-canonical elements may contain defined functions.

language construct. For instance, the `if` heating/cooling rules could be equivalently replaced by the following:

CONTEXT
`if` □ `then` — `else` —
[result(Bool)]

Such evaluation context statements must contain a single hole that indicates which subexpression is to be evaluated, whereas the syntactic sort deemed to be final (used within side conditions of the generated rules) is given as an attribute in square brackets.

In some situations, the expression in need of evaluation should not be rewritten according to the same rules used to handle the top of `k` but according to a different set of rules specific for the evaluation of the argument in question.

For example, suppose that expressions by default evaluate to their values. Within an assignment statement, however, the variable in the left-hand side should evaluate to an address, not value. Therefore, the context hole is wrapped so that a different, special set of rewriting rules would apply:

CONTEXT
$$\frac{\square}{\texttt{lvalue}\ (\square)} = —$$
[result(Addr)]

This context would then generate a heating/cooling pair:

RULE ASSIGN-HEAT
$$\frac{\bullet_K}{\texttt{lvalue}\ (L)} \curvearrowright \frac{L}{\square} = —$$
requires $\mathrm{isAddr}(L) \neq_K \mathrm{true}$

RULE ASSIGN-COOL
$$\frac{\texttt{lvalue}\ (L)}{\bullet_K} \curvearrowright \frac{\square}{L} = —$$
requires $\mathrm{isAddr}(L) =_K \mathrm{true}$

The `lvalue` wrapper is in this case responsible for transforming L into a value of sort Addr.

3 AGDA

Most strongly typed programming languages are based on a variant of the Hindley-Milner type system that includes only parametrised types, not dependent types.[1] Agda [11], on the other hand, is a strongly typed, pure, functional programming language with dependent types, based on a type theory à la Martin-Löf [24]. Due to the Curry-Howard correspondence that is to be explained in Section 3.2, Agda can also serve as an interactive theorem prover, allowing to prove mathematical theorems (in a constructive setting) and to check and run such proofs as algorithms.

Agda is the latest in the tradition of languages developed in the programming logic group at Chalmers University of Technology in Göteborg, Sweden, that dates back to 1983 [2]. The group of languages is known as the ALF family after its very first representative [21] (in temporal order). Some other Chalmers languages are Agda 1 [1] (a precursor to the current Agda system whose full name used to be 'Agda 2' before it replaced Agda 1 altogether) and Cayenne [7]. Dependently typed languages/proof assistants outside the Chalmers tradition are, e. g., Coq [8] and Idris [12].

Agda has been mainly developed by Ulf Norell with the goal of bridging the gap between the theoretical presentations of type theory and the requirements of a practical programming language [25]. The language comes with many practical features such as pattern matching over algebraic datatypes and type-safe treatment of metavariables but also with its Emacs interactive development mode.

The syntax of Agda is very plain with a very few special characters. In many aspects it is (superficially) reminiscent of Haskell including the fact that it is whitespace-sensitive: declarations are delimited by newlines with their scope indented. The most remarkable differences are support for mixfix operators (such as `if_then_else_`) and unicode identifiers.

For this reason, the next immediate section is already devoted to the question of why there is so much ado about dependent types at all.

Thompson's *Type Theory and Functional Programming* [34] in spite of its age is still a good book on the subject of dependent type theory and its connection to constructive mathematics. For a fully-fledged introduction to Agda, see the introductions of Bove and Dybjer [10] and Norell [26]. Finally, the versatileness of programming with dependent types is shown by Oury and Swierstra [27].

1 Haskell can simulate some aspects of dependent typing. [22]

3.1 DEPENDENT TYPES: A PRIMER

The key novelty that a functional programmer must grasp with regards to dependent types and their impact is the generalisation from simple function spaces $S \to T$ to *dependent* function spaces $(x : S) \to T$ (also known as Π-abstractions) where T may involve x. The crux of the culture shock lies precisely in this: a type can depend on a value that itself does not have to be a type. Hence it is possible to have types that refer not just to other types but also to numbers, strings, etc. Another aspect of this phenomenon is that when a function is applied, the *value* of the argument can be reflected in the resulting *type*.

Dependently typed programming can be also understood as programming with data that maintain invariants verified via typechecking. With a richer type system, more complex behaviours can be expressed compactly as types. Functions that inhabit those types are guaranteed to be correct with respect to the specified behaviours by virtue of having typechecked.

For the purposes of demonstration, we take the canonical example of lists (first without and then with internalised length invariants) and define some functions working with them.

3.1.1 *Parametrised Datatypes*

The following examples are part of a single literate Agda file that can be typechecked either with the official Agda system or with IKAgda. Each Agda file is required to have a top-level module – all the program's declarations go inside it.

We introduce a new datatype straight ahead:

Datatypes are written in GADT-style.

```
module DepStructs where

infixr 5 _::_
data List (A : Set) : Set where
  []    :                 List A
  _::_  : A → List A →    List A
```

Set is a universe, i. e., a type whose inhabitants can be used as types.[2] That constitutes the sole difference between what would otherwise be considered value-level expressions and types, since both values and types are expressions in Agda: only those expressions whose type is a universe can be themselves used as types of other expressions.

2 Obviously, if the typing relation should not fall prey to self-referential contradictions of Russell's kind, it must not admit Set to be its own type – it cannot hold that Set : Set. For this reason, the universe to which Set belongs is of a higher level, Set_1, which in turn belongs to Set_2, and so on *ad* (countable) *infinitum*. This organisation is called a predicative hierarchy of universes. In addition, Agda's hierarchy is non-cumulative: it does not follow from $A : \mathsf{Set}_\alpha$ that $A : \mathsf{Set}_\beta$ for some $\beta \geqslant \alpha$, each A inhabits only its particular universe.

The introduced datatype is List parametrised by another type A. Such a parametric declaration introduces a collection of datatypes each of which could be in principle declared separately.

List has two constructors [] (which creates a list of zero length) and _::_ (which adds an element at the beginning of another list; the underscores indicate that it is an infix operator while the **infixr** keyword says that it is right-associative with priority 5).

Since Agda as such comes with no base types, we have to define some ourselves in order to provide List with an actual parameter. What about Peano-style natural numbers?

```
data ℕ : Set where
  zero : ℕ
  succ : ℕ → ℕ
```

Now we can create a list of natural numbers like this:

```
[1,0,2] = one :: zero :: succ one :: []
  where one = succ zero
```

The example exploits the fact that basically any continuous string of characters counts as identifier in Agda. The name [1,0,2] that we just let denote the list-valued nullary function is as good as any other name that would not clash with a keyword or an already defined name.

3.1.2 *Functions*

The way of defining functions should be familiar to functional programmers. A major difference is that in Agda any pattern-matching function (a function with at least one argument listed in the left-hand side) must be equipped with an explicit type signature as dependent function spaces cannot be inferred automatically from the defining clauses [26]. Functions can be introduced either by λ-terms or clausally by left-hand-side patterns built up from variables and data constructors. Function applications β-reduce as usual.

If we wanted to define a function looking up an i-th element of a given list at this point, we would be forced to wrap its result into a polymorphic option type that encapsulates 'meaningful' results in addition with a 'null' value.[3]

The definition of the lookup function is virtually identical to its Haskell variant, it is defined recursively with the result wrapped within `Maybe`. With regard to the recursive part, we recognise two cases: a base case for where the position is zero and where we return the head of the list, and a step case where the position is matched

3 More formally, an option type is a tagged union of a unit type and the encapsulated type.

with succ i that leads to a recursive call of the lookup function on structurally smaller inputs.

```
data Maybe (A : Set) : Set where
  just    : A → Maybe A
  nothing : Maybe A

infixl 4 _!!_
_!!_ : {A : Set} → List A → ℕ → Maybe A
[]       !! _       = nothing
(x  :: _) !! zero    = just x
(_  :: xs) !! (succ i) = xs !! i
```

The first argument of _!!_, being enclosed inside a pair of curly brackets, is understood to be *implicit*. The implicit function space has the same meaning as the standard function space (with round brackets) except that when applying the _!!_ function the first argument (i. e., the type parameter A for List) does not have to be explicitly supplied and the typechecker will attempt to figure out its value. This effectively renders the function as (parametrically) polymorphic.

Should we need to provide a specific value for the implicit argument, we write the function in its prefix form and provide the value in curly brackets:

```
centre = _!!_ {ℕ} [1,0,2] zero
```

Metavariables are special meta-level variables not bound anywhere in the program.

To imitate the default automatic behaviour we could put a *metavariable* in place of the term that Agda should infer on its own:

```
centre' = _!!_ {_} [1,0,2] zero
```

The underscore represents a non-interactive, 'go figure' metavariable (as opposed to the other kinds of metavariables used during interactive development that we do not discuss here). Also note that an underscore occurring in the left-hand side works as a wildcard pattern or an anonymous variable that matches any input, not as an actual metavariable.

The disadvantage of _!!_ is that its user is always left with the burden of manual extraction of the actual result even when an empty list is certain to be never passed in. Such annoyances are unavoidable with lists because they do not contain the structure's length as part of their type information. Put simply, any function that accepts a List must cover all possible lengths of the input.

3.1.3 *Inductive Families*

With dependent types we can internalise the length information as an *index* of the datatype that consequently allows for the function's type to be more articulate and to exclude, say, empty structures. The Maybe

workaround is not needed anymore because the type itself demands that the function gets applied to non-empty structures exclusively.

```
infixr 5 _,_
data Vec (A : Set) : ℕ → Set where
  <>  :                          Vec A  zero
  _,_ : {n : ℕ} → A → Vec A n →  Vec A (succ n)
```

The declaration introduces an inductively defined *family* of datatypes indexed by the length. In addition to being parametrised like List (parameters are to the left of the colon in the type declaration), a vector is also indexed by a natural number (to the right of the colon) that encodes the length of the particular data structure.

For instance, one member of this family is the type Vec ℕ (succ zero). The expression zero , <> has this type. On the contrary, neither <> (its index is zero, not one) nor (just zero) , <> (it is parametrised by Maybe ℕ instead of plain ℕ) are expressions of that type.

The parameter differs from the vector in that the former is invariant and fixed while the latter varies and is defined locally per constructor so that all possible values for the index are targeted. It is not possible anymore to define each member of the inductive family independently of others as it was with collections of parametrised datatypes.

Now we may proceed to the definition of the lookup function for vectors whose safety is guaranteed by the type checker. First, we call the datatype of finite sets to our aid:

```
data Fin : ℕ → Set where
  fzero : {n : ℕ} → Fin (succ n)
  fsucc : {n : ℕ} → Fin n →  Fin (succ n)
```

Fin n contains exactly n elements. The idea is that the lookup function accepts only elements of a finite set $\{0, 1, \ldots, n-1\}$ as positions to be looked up in a vector of length n:

```
infixl 4 _[_]
_[_]  : {A : Set} {n : ℕ} → Vec A n → Fin n → A
<>       [ () ]
(x  ,  _) [ fzero ]   = x
(_  , xs) [ fsucc i ] = xs [ i ]
```

The first clause has the *absurd pattern* '()' to signify that the case is to be rejected by the type checker because no suitable constructors can be used (in this case, there are no constructors for Fin zero). Naturally, no right-hand side needs to be provided in such a case. A well-typed application of _[_] is thus guaranteed to involve a non-empty vector and to evaluate to an element[4] of the vector.

4 The type per se does not capture the fact that the result is the vector's i-th element – a function that ignores the position and always returns the first element is a term of that type, too.

3.1.4 *Inaccessible Patterns*

In the last function definition, the vector indicies were all conveniently implicit. If we tried to make it implicit, we would soon find out that the typechecker complains about a variable occurring multiple times in a left-hand side. Agda requires patterns to be *linear*, that is, the same variable must not occur more than once. This is not a big deal in languages without dependent types that can consider each of the arguments independently. Here, however, the issue of pattern non-linearity necessarily arises due to mutually related datatype indices. We lose the ability to match each argument separately since pattern-matching on one argument can enforce the value of other arguments by instantiation of datatype indices.

That is when *inaccessible patterns*, prefixed with a dot, come into play:

```
_[_]' : {A : Set} {n : ℕ} → Vec A n → Fin n → A
_[_]' {A} {zero}   <>       ()
_[_]' {A} {succ .n} (_,_ {n} x _)  fzero    = x
_[_]' {A} {succ .n} (_,_ {n} _ xs) (fsucc i) =
  _[_]' {A} {n} xs i
```

The accessible portion of the left-hand side must form a linear pattern. The inaccessible parts are used only for typechecking; they are discarded at run time during actual pattern matching as they are certain to match once they have typechecked [10].

3.2 PROPOSITIONS AS TYPES

A series of observations made by Haskell Curry and William Howard from the 1930's until the mid-twentieth century led to establishing a link between the typed λ-calculus (i. e., a model of computation) and constructive propositional logic (i. e., a proof system), which is a logic that does not admit the law of the excluded middle as an axiom. Under the isomorphism, types are in one-to-one correspondence with propositions and terms of those types with proofs of those propositions. It is sometimes called the propositions-as-types principle.

Dependent types were introduced in order to extend the correspondence from propositional to predicate logic. After further work, the correspondence became the building block for Martin-Löf's intuitionistic type theory [24] which in turn represents the origins of the modern development of dependently typed programming languages.

The logic view as opposed to the programming view is summarised in Figure 1. Each proposition or predicate formula is interpreted as a set of its proofs. For example, a conjunction of two propositions is identified with a cartesian product of the two sets of proofs: all proofs of $A \wedge B$ have the form $\langle a, b \rangle$ where a is the proof of A and b is a proof of B.

Logic	Programming
(true) formula	(inhabited) type
proof of a formula	term of a type
universal quantification	dependent function space
existential quantification	dependent pair
conjunction	cartesian product
disjunction	disjoint union
implication	function type
negation	function type to the uninhabited type
modus ponens	function application
provability of formula	inhabitation of type
proof checking	type checking
interactive theorem proving	interactive term construction

Figure 1: Summary of the Curry-Howard correspondence.

Any programming language with dependent types can thus model the constructive predicate calculus. Given that typechecking is decidable (as discussed for Agda in the next section), any such language can be used as a theorem prover under the Curry-Howard correspondence.

To make it more clear, let us prove a simple theorem in Agda. We will show that zero is a right identity element with respect to addition of natural numbers.

First, we define the addition itself by primitive recursion in the first argument:

```
infixl 6 _+_
_+_ : ℕ → ℕ → ℕ
zero    + m = m
(succ n) + m = succ (n + m)
```

Further we need a means to reason about equality within Agda types because we cannot directly prove any properties of Agda's built-in definitional equality:[5]

```
infix 4 _≡_
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

`_≡_` is the diagonal relation on A. In set theory, it would be defined as the least binary relation on a set A such that $x \equiv x$ for all $x \in A$. In type-theoretic terms it is a family of proofs of "being *propositionally* equal to x" that is inhabited only when the index is *definitionally* equal to x; `refl` is the witness that the two terms of type A are equal. We have

5 The internal notion of equality views those terms as equal that normalise to the same value [5]. Normalisation involves β-reduction and rewriting according to function clauses (by pattern matching). Fast-forward to Section 4.5 for more technical details.

thus exposed the internal notion of equality so that it is amenable to be a component of types/propositions.

The equality relation is also a congruence in the sense that it is compatible with unary functions:

```
cong : {A B : Set} {x y : A} → (f : A → B) → x ≡ y → f x ≡ f y
cong _ refl = refl
```

Beware that refl in the left-hand side is not identical to the constructor of the same name in the right-hand side. What we match on is the witness that x and y are equal, whereas refl to the right is the witness of $f\ x \equiv f\ y$. This constellation is certain to typecheck thanks to the fact that $x \equiv y$ (which is proven by the input witness) and the fact that f is a function (any function gives the same value whenever it is applied to the same/equal argument).

The type can be read as $\forall n \in \mathbb{N} . n + 0 \equiv n$.

Now that we have everything prepared we can come to grips with the desired theorem:

```
thm : (n : ℕ) → n + zero ≡ n
thm  zero     = refl
thm (succ n)  = cong succ (thm n)
```

The proof is by induction on n. The base case is most simple: zero + zero reduces to zero and so all we need to prove is that zero ≡ zero.

In the step case, we get succ (n + zero) after reduction per the second clause of _+_. We then rely on the hypothesis that the theorem holds for the structurally smaller number, namely n, to build up a proof of succ (n + zero) ≡ succ n using the congruence property.

Note that the theorem does not take advantage of the way in which addition is defined. On the other hand, if the theorem was formulated with zero + n ≡ n, the function application would immediately get reduced in accordance with the first clause of _+_. The proof thereof would be trivial, too:

```
thmEasy : (n : ℕ) → zero + n ≡ n
thmEasy _  = refl
```

Both theorems show that Agda can and does normalise *open* expressions (containing free variables) during typechecking. In non-dependent functional languages, this kind of normalisation is performed only during proper execution of the program (their types do not involve usual function applications) and even then only to simplify *closed* expressions [10, sec. 5.2].

All in all, programming really ends up to be proving.

3.3 DECIDABILITY OF TYPECHECKING

The price paid for the expressivity of dependent types is the complexity of their typechecking. Since any expression can occur as part of a type and types in turn need to be evaluated or normalised in order to determine their equality, computation routinely occurs during

typechecking with dependent types. For the typechecking to be decidable it is therefore necessary to guarantee that all computations eventually terminate. In other words, the underlying term language must be strongly normalising.

Because of this, all functions in Agda programs are required be total: they must be terminating and defined on all possible combinations of inputs. Obviously, due to the undecidability of the halting problem, it is not possible to precisely delineate the set of total functions. Even if necessarily incomplete, the solution can be conservative: if a function is accepted as total it is sure to be so but some of those rejected can be in fact total.[6]

The official Agda typechecker takes this approach and enforces totality by three relatively simple yet conservative checks [19]:

- function clauses must cover the whole domain of the function space;
- recursion is restricted to calls with structurally smaller arguments;
- data constructors must be strictly positive.

The *strict positivity* condition rules out declarations in which a datatype lies to the left of an arrow in the type of its constructor's argument. The ubiquitous example is the datatype Bad that gets rejected by the typechecker:

```
data Bad : Set where
  bad : (Bad → SomeType) → Bad
```

One of the consequences of the non-termination checks is that Agda has uninhabited types for which there is absolutely no way to construct values. In Haskell, for instance, which allows possibly non-terminating general recursive functions, non-strictly positive datatypes and also has the `undefined` constant, all types are inhabited because a non-terminating function inhabits any type.

A common use of a datatype with no elements is the trivially false proposition:

```
data ⊥ : Set where
```

Note that if every type was inhabited, it would by Curry-Howard mean that every formula is provable, which is the very definition of an inconsistent theory in logic. A type system in which all types are inhabited cannot therefore correspond to a consistent logic.

6 Termination checking for such false negatives can be deactivated on a case-by-case basis by the pragmas `{-# OPTIONS NON_TERMINATING #-}` and `{-# OPTIONS NO_TERMINATION_CHECK #-}`.

4

A 𝕂 SEMANTICS OF AGDA

The task of giving a formal semantics to any programming language is intrinsically concerned with assigning exact meaning (semantics) to syntactic constructs of the language. In order to avoid ambiguities of ordinary human language, the result needs to be formulated within a mathematically rigorous formal system. The 𝕂 framework further extends the meaning of formal descriptions and derives software tools from them in an automated way.

Agda as such has no formal semantic definition besides this work. Much of the semantics development is thus driven and informed by sources that are not specifically intended to this end. The source material consists of the following:

- articles concerning semantics of particular type theories such as those discussed in the next section or the tutorial by Löh et al. [20];
- the reference manual available at the Agda wiki [3];
- a great variety of tutorials and real-world code written in Agda;
- the code base, tests and examples distributed with the official Agda typechecker.

Any effort to give a formal semantics of Agda in the present state necessarily goes hand in hand with the process of synthesising information acquired from these sources. Moreover, properties pertaining to the official implementation of Agda rather than Agda as a language need to be distinguished. Validity of the semantics is primarily verified by testing for correspondence of behaviour with the official typechecker, which makes the tools automatically generated by 𝕂 indispensable.

The situation is quite similar to the one faced by the developers of the 𝕂 PHP semantics; my attitude was influenced by theirs and so their own explanation deserves to be quoted in full [15, sec. 1] (with references omitted):

> "Some programming languages, such as Standard ML, already come with a formal specification. Others, such as C and JavaScript are specified in English prose with varying degrees of rigour and precision, and have recently been formalised. PHP is only implicitly defined by its *de facto* reference implementation (the Zend Engine), and the (informal) PHP reference manual. Due to the lack of a document providing a precise specification of PHP, defining its formal semantics is particularly challenging. We

> have to rely on the approximate information available online, and a substantial amount of testing against the reference language implementation. We do not to [*sic*] base our semantics on the source code of the Zend Engine in order to avoid bias towards inessential implementation choices."

This chapter presents an executable semantics of Agda written in $\mathbb{K}$ as well as its major components and some key decisions made during its development. The next section is devoted to a possible alternative approach to the problem of Agda semantics, putting the work into a broader perspective. I continue with details on parsing the concrete syntax of Agda programs in relation to $\mathbb{K}$Agda. The rest of the chapter covers individual constructs of the Agda language, normalisation and the dependent type system (i. e., the typechecking and type inference algorithms) as they are realised in $\mathbb{K}$Agda.

4.1 CORE TYPE THEORIES

Elaboration usually involves desugaring and annotation of untyped expressions with types.

It is a common practice when implementing typecheckers for functional languages to *elaborate* the concrete high-level syntax into a small yet sufficiently expressive typed intermediate language, i. e., a *core type theory*.[1] Among the advantages of such an approach is that the core theory, unlike the 'verbose' source language, is well-understood, is easily reasoned about, comes with a trusted code base, and, most importantly for the topic at hand, very often has an explicitly defined formal semantics.

The official Agda typechecker, on the contrary, does not utilise elaboration. The extensions to UTT_Σ, which is considered as Agda's core type theory in Norell [25, sec. 1.3], are implemented directly, rather than in terms of primitive constructs.

This is not utterly unreasonable: the method of elaboration comes with a price, and thus it is not always preferable to have recourse to it. For example, one of the consequences of elaboration is that errors get detected in code that is not actually given by the user, which may cause significant difficulties for error reporting. A key feature of Agda– its interactive development mode – is similarly affected since it depends on code-passing between the user and the typechecker.

Furthermore, Agda is designed with emphasis on practical use. For a practical language the 'user-friendliness' of a feature is important than its theoretical foundation, which suggests that at times it could be beneficial to introduce a new feature even if its elaboration was not possible, hence undermining the concept of having a core theory at all. (Too frequent modifications of the core language also go against

1 For instance, Haskell's GHC internally translates into System F_C [33], while Idris, a language even more akin to Agda than Haskell, is elaborated to a dependently typed λ-calculus $\top\top$ [12].

its being trusted and well-understood.) Some useful features may not admit straightforward theoretical formulation whatsoever.

Still, there have been attempts to determine precisely a core language into which Agda could be elaborated. Specifically, Agda developer Andreas Abel proposed in his DTP 2011 talk [4] to derive a core for Agda from the dependently typed core language ΠΣ [6]. Another effort in this direction is Mini-TT by Thierry Coquand et al. [13]. To my knowledge, none of these attempts allow for Agda to be completely elaborated, mainly due to its various advanced features such as metavariables, implicit arguments or even pattern matching. A theory that allows for representation of all these features runs the risk of being overly complicated, thereby not fulfilling its purpose as the very *raison d'être* of elaboration is to simplify the process of typechecking.

The 𝕂 approach to semantics definitions seems to solve these problems quite conveniently. With 𝕂, rewriting rules for the language constructs get specified *directly*, while retaining the potential for theoretical analysis within the broader context of the rewriting logic [23]. It ceases to be necessary to impose restrictions on the language itself so that it can be expressed in the core theory under all circumstances – it becomes possible to convey exact meaning of all language constructs, including the more advanced or practical ones, with no need of elaboration.

4.2 SYNTAX

4.2.1 *Parsing and Scope Analysis*

As part of the `kompile` process, 𝕂 generates a parser based on the syntax module that contains the grammar of the programming language. The resulting parser transforms programs into 𝕂 abstract syntax tree (AST) format [9] that is consequently accepted by other 𝕂 tools such as `krun`.

Given that the primary focus of 𝕂 is semantics, it is not surprising that 𝕂's capabilities to describe syntax are sometimes insufficient. In the case of Agda, an issue one encounters virtually immediately is the impossibility to determine how to treat whitespace. This is a problem because the layout of Agda programs must be examined in order to delineate declarations and their scope. The parser generated by 𝕂, however, cannot be instructed about interpreting the whitespace characters: they are always viewed as mere token separators.

The 𝕂 semantics of the whitespace-sensitive language Python [17] employs an external parser, too...

In anticipation of such situations, 𝕂 tools can be provided with an external, user-specified parser in lieu of the default, automatically generated one. To this end, I have adapted the official Agda typechecker and its built-in parser so that it produces 𝕂 AST. The

```
one = succ zero
inc = λ x → x + one
```

(a) The input for the external parser.

```
one_name = succ_name zero_name
inc_name =
  λ x_var → ((_+_name x_var) one_name)
```

(b) The output of the external parser.

Figure 2: Scope analysis and infix/mixfix operator resolving.

modifications introduce a new type class `KShow`, an analogue of the `Pretty` class that the typechecker uses for pretty-printing.

The official Agda typechecker does not internally operate on concrete Agda code as the programmer writes it. Instead, the *concrete* syntax is translated into its *abstract* representation by performing scope analysis, resolving infix operator applications and additional minor desugaring: the typechecker then works with these abstract structures. Should a need to pretty-print arise afterwards – say, an error occurs – a reverse translation is carried out, i. e., translation from the abstract syntax to its concrete counterpart. The result is a (slightly cleaner) variant of the programmer's original input, recognisable enough to aid with debugging.

With respect to syntax, I take advantage of as much functionality of the official typechecker as possible. Some aspects of the concrete-to-abstract translation would belong to the desired functionality allowing IKAgda to skip these aspects of scope analysis etc. Yet, constructing the final IK AST out of the abstract syntax directly is not an option since the syntax is too tied in with the internal mechanisms of the official typechecker. Too much bias would be brought in if the abstract representation was taken as input to IKAgda, which is to be an *autonomous* formalisation of Agda's semantics after all.

Thus, a dilemma enters the stage: on the one hand it is appropriate to translate from concrete to abstract while on the other hand the concrete syntax ought to remain the base for the output of the external parser. The answer is surprisingly simple. Just translate back from abstract to concrete!

This certainly sounds like codswallop to the attentive reader, so an explanation is in order. The point is that the subsequent, abstract-to-concrete translation has been altered: even though the scope data are still dropped, some portions of them get embedded in the concrete syntax. The concrete syntax obtained after the translations is thus substantially different from the original concrete syntax and contains original information. It now stores 'hard' names (i. e., identifiers of functions, datatypes, modules, etc.) and variables separately. Also, the infix/mixfix operators are guaranteed to be in their prefix form, which does not hold for the original concrete syntax.

The scope analyser distinguishes between the kinds of names all of which are at first parsed simply as names with no distinction.

Figure 2 shows the effect of scope analysis and operator resolving that the external parser performs in addition to the parsing. The iden-

```
module Id where
  id : (A : Set) → A → A
  id = λ A x → x
```

(a) A simple Agda module.

```
module IdDesugared where
  id : (A : Set) → (v : A) → A
  id = λ (A : _) → λ (x : _) → x
```

(b) The same module, desugared.

Figure 3: Desugaring of the generic identity function.

tifiers that were originally undifferentiated are now subscripted as the scope analysis has matched them to their declarations (in which case they are annotated as names) or bindings (variables). The infix notation of the addition function is also converted into its prefix form.

To be more precise then, the dilemma is solved by extending the concrete syntax in such a way that it can keep some useful bits otherwise characteristic for the abstract representation. When this extended concrete syntax is `KShow`-printed, the output is the 𝕂 AST, which is in turn the input for 𝕂Agda itself.

4.2.2 *Desugaring*

Some language constructs can be equivalently formulated in terms of others, of which the former is just a particular application. The reason for having such redundant language constructs at all is to protect the programmer's sanity because the more general formulations tend to be quite verbose. Besides, the syntactic sugar may be most worthwhile with in regard to readability and maintainability of programs.

When giving semantics, though, no redundancy is welcome. All non-fundamental constructs are therefore desugared in order to reduce the variability of cases that have to be covered explicitly in the semantics.

None of the changes made to the official typechecker simplify the syntax in preparation for easier processing. With the exception of some that are performed by the official type checker itself during the concrete-to-abstract conversion, all such behaviour-preserving transformations are done exclusively by means of 𝕂 macros.

The most notable of these transformations are:

- annotation of untyped (domain-free) bindings with metavariables (Section 3.1.2) playing the rôle of types, and
- conversion of non-dependent function spaces into dependent ones.

Their purpose is to allow the actual semantics to be concerned only with the general form of typed bindings, ignoring the syntactic sugar altogether. Figure 3 gives an example in which the code (b) is derived from (a). The name v is an instance of a fresh variable, it might as well

be a w, ω, or any other name that would not capture or be captured. The underscores in the typed bindings are metavariables for types that should be automatically inferred.

An effect to be especially stressed is that all λ-abstractions, as they occur in IKAgda, are annotated with explicit type labels. Even though such an approach may cause unnecessary duplication of type information, it is outweighed by the gained capability to type the abstractions directly.

Other semantics, e.g. [9], manipulate the AST nodes directly and therefore their rules cannot be followed so easily.

The macros along with the syntax of IKAgda in general strive to imitate Agda's own syntax to the maximum degree, which in effect facilitates comprehension of the semantic rules, at the very least for those familiar with regular Agda code.

4.3 SEMANTICS: AN ENTRÉE

The tutorials [30] bundled with IK Framework dealing with type systems have had a great impact on the development of IKAgda. The type systems there are all non-dependent, though, and so they can afford to forget about an expression once its type has been determined. Not so with Agda. In a dependent setting it is still necessary to remember the expression even after it has been typed since type of a (dependently typed) function may very well depend on the *value* of the argument. Rather than keeping the type exclusively at the expense of the value, IKAgda has to consider both of them as the result of typing.

The typing procedure thus yields a pair, the notation for which is $\langle \mathsf{E} \downarrow \mathsf{T} \rangle$. Its meaning is that the type that has been inferred for the expression E is T.

A related important idea – shared among most of IKAgda operations – is that a problem can be solved by analysing as much information as extractable from the components of its input. Therefore, context statements (recall from Section 2.3) are often used to infer and pre-process the types of the components.

One way in which the type inference result is inspected is to guarantee that an expression can serve as a type of another expression, i. e., that *its* type is a universe. This need is so commonplace that it was generalised as a separate evaluation context called `checkSetType`. If the context evaluates successfully, the type-inference pair obtained from it is surely of the form $\mathtt{Set}_\alpha$ where α is a universe level. Rules may further inquire about the universe level (as with the pattern in Figure 6).

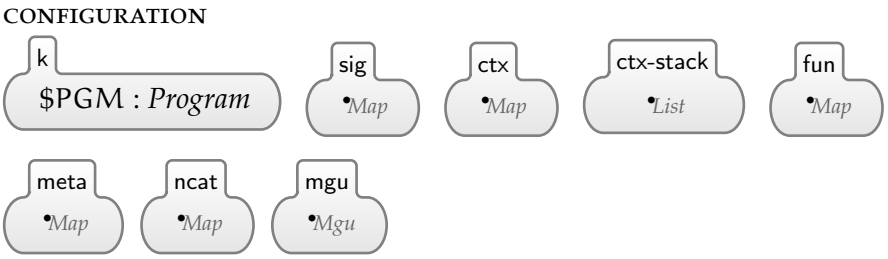


Figure 4: IKAgda configuration.

4.3.1 *Configuration*

The IKAgda configuration (Figure 4) consists of eight cells in total:

1. `k` is the cell in which the rewriting primarily takes place as explained in Section 2.1. The cell is initialised with the parsed input program.

2. `sig` is the map of type signatures, assigning types to globally defined datatypes, constructors and functions.

3. `ctx` is the typing context, i. e., a map with variables as keys and their respective types as values.

4. `ctx-stack` is an auxiliary cell storing a list of the typing contexts so that they can be later restored after the construct being processed is dealt with. The cell works as a stack with `saveCtx` and `loadCtx` as its push and pop operations.

5. `fun` maps function names to their bodies, which can be either usual expressions (λ-abstractions) or, if the function is spread over multiple clauses, a `'fc`-list of the clauses (see page 30).

6. `meta` contains metavariables (covered in Section 3.1.2) and their types. In contrast with `ctx` and `sig`, the values are not merely types but a pair of an expression and its type. The idea is to monitor the value of the metavariable, not just the type, as it changes through imposition of unification constraints.

7. `ncat` associates general `Names` with their specific category. The external parser distinguishes only between variable and non-variable names, cf. page 23. In the semantics, however, it is appropriate to distinguish among the categories of the non-variables themselves. The map thus contains items of the form:

   - `Name`(N) ↦ `Data`(N), where N is an identifier of a type constructor (e. g., List);
   - `Name`(N) ↦ `Con`(N) for data constructors (e. g., _::_);
   - `Name`(N) ↦ `Fun`(N) for functions (e. g., _!!_).

After a name is added to the name-category map `ncat`, any general `Name` occurrences of it are replaced with the more specific name that indicates its category.

8. `mgu` is an acronym for 'most general unifier'. In this cell, the various unification constraints are accumulated.

The configuration determines the state during execution. The `k` cell gets repeatedly rewritten as long as possible with three possible outcomes:

A. either reaching $\bullet_K$ that signifies that the program has been wholly consumed, which is a necessary but not sufficient condition of a completely successful run;

B. or getting stuck if, at some point, no rules match the current configuration, implying there is an error in the program or in the semantics itself;

C. or looping indefinitely, e. g., when one rule produces a configuration that is matched by another that in turn produces a configuration that is matched by the first one.

4.4 DECLARATIONS

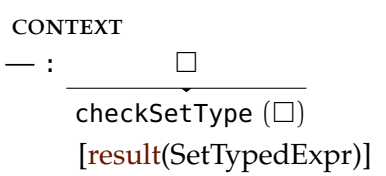
Declarations in 𝕂Agda are all processed in accordance with the scheme:

1. The declaration is checked to be well-formed, namely:
   - that all its variables are bound (guaranteed by Agda's scope analyser from Section 4.2.1),
   - that those constituents that are supposed to be types really belong to a universe, and
   - that conditions pertaining to the particular kind of declaration are satisfied.
2. The configuration cells are updated accordingly and the rest of the program is treated with reference to this new configuration.

The most elementary declaration is a type signature for a function, which comes with virtually no additions to the two steps mentioned in the scheme, cf. Figure 5. First, the expression after the colon purported to be a type is checked to ensure that it is an actual type, i. e., that it belongs to a universe. If the check passes, the signature itself gets consumed while the new item is added to the signature map. The `renameBoundVariables` function that replaces bound variables with fresh variables is used to prevent name clashes during unification.

The other kinds of declarations handled in 𝕂Agda – datatypes and functions (as in a list of function clauses without the signature) – are described in the rest of this section.

1.



2.

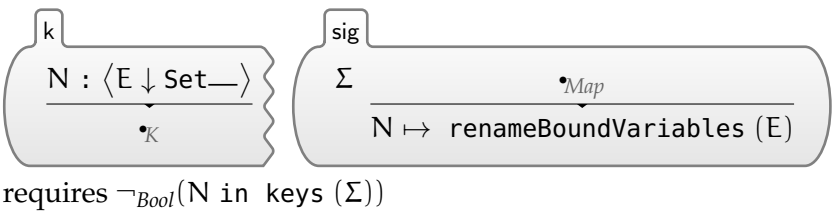


Figure 5: IKAgda implementation of adding of a function type signature.

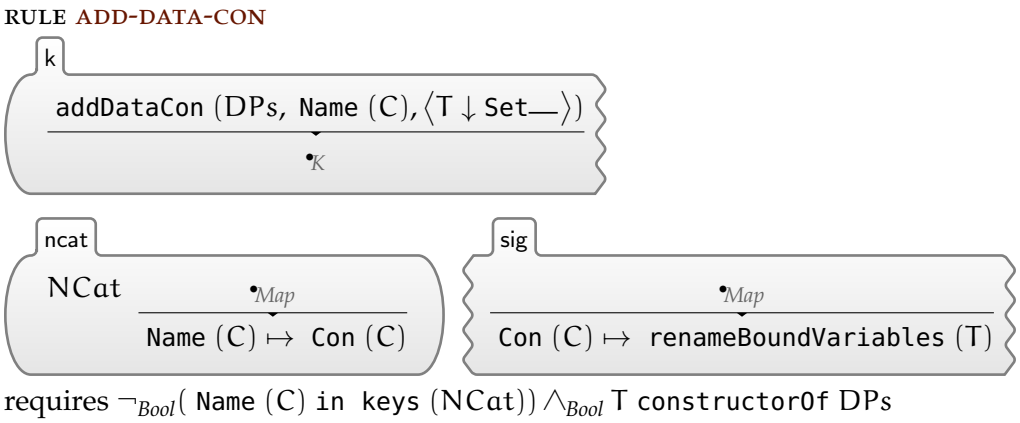


Figure 6: IKAgda implementation of adding of a data constructor's signature.

4.4.1 *Datatypes*

The ability to define custom datatypes is a central feature of Agda, or of any dependently typed language for that matter. In Agda, datatypes are introduced by a **data** declaration consisting of the new type's name, the list of its parameters, function space and the list of its constructors with their respective type signatures.

During processing of a **data** declaration, the types of the constructors are checked for being genuine types, i. e., if they, after being prepended with the datatype's parameters, do typecheck against a universe. The constructors are similarly checked for being constructors of the type they claim to be. The actual adding of a data constructor is then shown in Figure 6.

If everything goes well, the `sig` and the `con` cells are populated: the former with the checked types of the constructors and the latter with

```
data Vec (A : Set) : ℕ → Set where
  <>  : Vec A zero
  _,_ : {n : ℕ} → A → Vec A n → Vec A (succ n)
```

(a) Datatype Vec.

```
Vec  : Set → ℕ → Set
<>   : {A : Set} → Vec A zero
_,_  : {A : Set} → {n : ℕ} → A → Vec A n → Vec A (succ n)
```

(b) Vec-derived constructors.

Figure 7: Type signatures derived from a **data** declaration.

the list of data constructors associated to the datatype they construct. Signatures similar to those of Figure 7 would be added to `sig`.

Inductively defined families of datatypes constitute just a special case of this approach towards **data** declarations. They are likewise only prepended with the datatype's parameters and added to the configuration cells with the sole difference that the base function space of type constructors for inductive families would be non-trivial.

4.4.2 *Function Clauses*

Functions are defined as a block of function clauses with each left-hand side starting with the function name, followed by patterns that the actual arguments are supposed to match.

As in Haskell, the order of the clauses matters and must be remembered (they are tried from top to bottom). In IKAgda they are stored in an ordered list of function clauses wrapped in the `'fc` label. After processing a function clause declaration the clause is stored in the `fun` map: either by initialising a new list of clauses or by updating a pre-existent one.

When a function clause is encountered, the type of the left-hand side of the clause is inferred by means of the rules of the type theory. The left-hand side is thus interpreted simply as an ordinary function application. The function that has its type from the type signature is gradually applied to the patterns of the left-hand side and the types of the formed applications attempted to be typed. At the same time, the types of the automatically typed variables are spelled out as the rôle of arguments forces constraints upon them. It is important to note that the patterns do not get explicitly checked against their types from the function's type signature.

Provided that the left-hand side has been typed successfully, the type of the right-hand side is inferred. The types of the left-hand and right-hand sides are checked to be unifiable against the target type of the function as well as against each other. If one of the checks fail, the

actual return type of the function and the function's signature are not consistent, and the IK rewriting cannot continue.

At last, the function clause is appended to the list of clauses associated with the function. It is stored in its basically original form, including the inaccessible patterns. Some wildcard patterns standing for implicit arguments inserted. A more important modification consists in that the function name in the left-hand side is replaced with a fresh variable for the purposes of pattern matching (to be explained in Section 4.5.1).

Also note that in its current state IKAgda does not check whether absurd patterns (marked '()') are really absurd – they are simply ignored, not even included among the list of a function's clauses. The proper approach would be to verify that there truly are no suitable constructors to be used in the context.

4.4.3 *Local Definitions*

There are two ways of declaring a local definition in Agda: **let** and **where**. The difference between them is that a **let** binding is a standard expression like a λ-abstraction but a **where** binding is a special language construct attached to a function clause declaration. Both are supported by IKAgda.

In order to support the local definitions, the syntax of expressions had to be extended so that it contain the local definitions affixed to the expressions that refer to them. The local definitions are not stored in a separate cell of the configuration.

Before a local definition is processed, IKAgda saves the names already present in the `sig` map. The local definitions are then propagated to the top of the `k` cell to be processed as though they were global definitions, updating the `sig` and `fun` cells of the configuration as per usual. The newly introduced names are afterwards filtered out by way of comparison with the saved names. The new names with their signatures and values are then incorporated in the expressions that refer to them using the extended syntax.

Configuration data that were created by the local definitions are discarded. At the end, the configuration holds exclusively global information whereas the local data are stored as annotations of the related expressions.

4.4.4 *A Note on Deficient Checks*

IKAgda does not recognise when one function's clauses should have been completely listed, it simply consumes any function clause with no regard to what declarations occurred between the present and the last clause belonging to the function. In other words, IKAgda typechecks each clause individually and not as components of a block.

This simple approach gives rise to certain issues. Insofar as these issues pertain to scope analysis – which, as a matter of fact, the described issue of intermingling clauses does – they get caught during the 'parsing' stage (see Section 4.2.1), not reaching the very permissive IKAgda. Still, situations like unreachable classes (e. g., due to the same constructor pattern's having been used twice) that trigger an error in the official type checker are not detected as faulty in IKAgda.

In addition, Agda is a total language, i. e., all functions are required to be terminating and their left-hand sides to cover all possible cases that are correct with respect to the arguments' types. However, IKAgda itself currently does not perform any coverage or termination checks, they are left out entirely.

The totality of Agda further implies that non-strictly-positive datatype declarations must be ruled out but again no checks for strict positivity are done by IKAgda. In particular, the Bad example of Section 3.3 is (wrongly) accepted by IKAgda without any issues.

4.5 NORMALISATION

Many of the IK rules working with typed expressions presuppose that the types are in weak head normal form `whnf` or normal form `nf`.[2]

Normalisation is implemented in IKAgda by means of IK contexts (explained in Section 2.3). It means that syntactic sorts `Nf` and `Whnf` (both subsorts of the sort of expressions) are defined to characterise the final evaluation results of the contexts.

The `whnf` and `nf` contexts both perform normalisation by β-reduction on the expressions as such, in an untyped way. The reduction relation is understood as usual, that is,

$$(\lambda x.s)t \to_\beta s[t/x],$$

where $s[t/x]$ denotes the standard, capture avoiding substitution of t for x in s.

Even though the same reduction technique is used in both `whnf` and `nf`, the difference between the contexts lies in the choice of subexpressions that get reduced. In `whnf`, only the subexpressions along the *spine* of the expression, i. e., the leftmost branch of its applications, need to be reduced, whereas the other subexpressions can be left unreduced. On the other hand, in a normal form all subexpressions are reduced.

A related yet still more mundane task the `whnf`/`nf` contexts are charged with is to replace occurrences of function names with their bodies according to the `fun` map. A function name (as opposed to names of datatypes or constructors) is *not* considered to be in normal form; it must be substituted with its body that gets further normalised as any other expression if necessary.

2 Meaning β-(weak head) normal forms, not βη-(weak head) normal forms.

```
_∨_ : Bool → Bool → Bool
false ∨ false = false
x     ∨ y     = true
```

(a) An Agda definition of disjunction.

```
_∨_ : Bool → Bool → Bool
false ∨ false = false
false ∨ true  = true
true  ∨ false = true
true  ∨ true  = true
```

(b) After expansion of (a).

Figure 8: An illustration of expansion of overlapping function clauses, performed internally in the official Agda typechecker [5].

If the function body is not a plain λ-abstraction but has multiple clauses, the `'fc`-list found in `fun` is wrapped in a `FunClauses` together with the relevant function name (to facilitate reverse type inference) along with the actual clauses. The container belongs to the sort of expressions and is considered to be in a normal form if it is not applied to a sufficient number of arguments or the match is found to be inconclusive as explained in the next section. That is to say, `FunClauses` behave as a function except that is reduced by pattern matching instead of β-reduction.

4.5.1 *Pattern Matching*

While other implementations of dependent type theory elaborate dependent pattern matching in terms of more fundamental elimination operators [16], Agda instead treats pattern matching as a primitive. In the official typechecker, the objective of pattern matching is to construct an effective case tree for each pattern-matching function based on its clauses [25, sec. 2.2]. If possible, overlapping clauses are transformed into equations that constitute definitional equalities within the type theory during the construction process, as depicted in Figure 8. Such a tree is then traversed during pattern matching in order to determine which right-hand side of the function to return.

As hinted at in Section 4.4.2, though, 𝕀KAgda does not construct case trees out of function clauses and stores them virtually identically to Figure 8a. Instead, the last step of processing a function clause is to append the clause (with minor modification to how it was given by the programmer) to the list of function clauses associated with the function, which is feasible because 𝕀KAgda takes a simple, operational view on pattern matching: it just attempts to match the actual application expressions with the required number of arguments against the left-hand sides of the stored clauses one by one.

As with unification, the simplified pattern matching is implemented using 𝕀K's built-in modules.

To allow for pattern-matching functions to occur in expressions, the clauses are stored in the `FunClauses` container. Pattern matching requires the left-hand sides of the clauses to be linear (Section 3.1.4) and therefore the list in `FunClauses` cannot store inaccessible patterns

in their original form suitable for typechecking rather than pattern matching. When constructing the `FunClauses` object the inaccessible patterns are hence replaced with wildcard patterns so that no variable occurs more than once in the left-hand side as desired.

The left-hand side of the first `FunClauses` clause is matched against the whole application expression with all arguments taken together. Two possibilities arise:

A. There is a mismatch in the sense that this clause's left-hand side cannot be unified with the given arguments at all. For example, with the first clause of `_∨_` from Figure 8a the application `false ∨ true` would result in a definitive mismatch that cannot be resolved. The next `FunClauses` clause is tried in such a case.

B. There is an inconclusive match. The arguments could be unified with the left-hand side but are too underspecified, for example as in `false ∨ neut` where *neut* is a variable, the first clause could still match. The normalisation of the function application is then suspended and the expression is preserved in its unnormalised form, since an inconclusive application of `FunClauses` is defined as a special normal form. Once the arguments become more concrete through imposition of constraints during typechecking, the application will reduce.

4.5.2 *Phase Distinction*

Most statically typed programming languages perform typechecking prior to and independently of the actual execution phase and are therefore said to respect the *phase distinction* [29, sec. 8.4]. In this sense dependently typed languages may be said to violate the phase distinction since they may require some normalisation to take place during typechecking (as discussed in Section 3.3).

However, the way in which programs are executed at runtime (closed expressions only, after type erasure, highly optimised) is distinguished from the way they are executed during typechecking (reduction of open expressions, with custom datatypes) even in dependently typed settings. There is indeed no distinction between terms and types *qua* syntactic categories yet the two execution models are still maintained to address their specific purposes and concerns – although now operating on the same underlying term language.

𝕀KAgda does not implement a genuine runtime execution model. All normalisation is performed only in order to acquire a (weak head) normal form of types. If an expression never occurs within a type, it is likewise never an object of normalisation. Both functions and types are stored unnormalised after processing their declaration.

$$\frac{\Gamma \vdash e_1 \downarrow S_1 \rightsquigarrow T_1 \qquad \Gamma, x : T_1 \vdash e_2 \downarrow S_2 \rightsquigarrow T_2}{S_1 \to_{whnf} \mathtt{Set}_\alpha \qquad S_2 \to_{whnf} \mathtt{Set}_\beta \qquad \Gamma \vdash (x : e_1) \to e_2 \downarrow \mathtt{Set}_{\alpha \sqcup \beta} \rightsquigarrow (x : T_1) \to T_2}$$

Figure 9: Type inference rule for Π-abstractions as presented by Norell [25, p. 20].

4.6 TYPE SYSTEM

The foundation of Agda is a dependent type theory. Still, as discussed in Section 4.1, Agda cannot be properly said to be based on a specific core theory. A type theory relates to Agda only to the extent it models Agda's basic functionality and serves as a groundwork on top of which Agda's peculiar features are developed.

It follows that in such a context the particular choice of type theory is not crucial but during the development of IKAgda I have still tried to remain faithful to the theory chosen by Norell [25]. The reader may wish to contrast the IK rules described below with the type inference and checking rules in Section 1.4 of Norell's thesis, Figures 1.4 and 1.5 in particular.

4.6.1 *Type Inference*

Upon encountering an untyped expression, IKAgda attempts to infer its most general type in accordance with the available[3] types of its subexpressions. To expose the manner in which the type theory is realised in IKAgda, I first discuss IKAgda implementation of type inference for dependent function spaces or, equivalently, Π-abstractions, and then more briefly the inference procedure for function applications.

The original type-theoretic inference rule for Π-abstractions can be seen in Figure 9. The notation $\Gamma \vdash e \downarrow T \rightsquigarrow t$ reads as "the type of e has been inferred as T with resulting term t". Notice how the judgement $\Gamma \vdash T_i : Set_\alpha$ is verified to determine whether T_i can be used to type the bound variable in the final expression. ($\alpha \sqcup \beta$ stands for the maximum of α and β.)

The IK rules based on the type-theoretic rule are then themselves displayed in Figure 10. The final type is obtained in four rewriting rules invoked in succession.

First of all it is necessary to verify that the typed binding is valid, which is assured by the contextual evaluation of `checkSetType` that checks that the type of the expression is a universe (1st stage). If the context has evaluated successfully, we may safely bind the variable

3 Including both previously automatically inferred types and types provided by the programmer via type signatures etc.

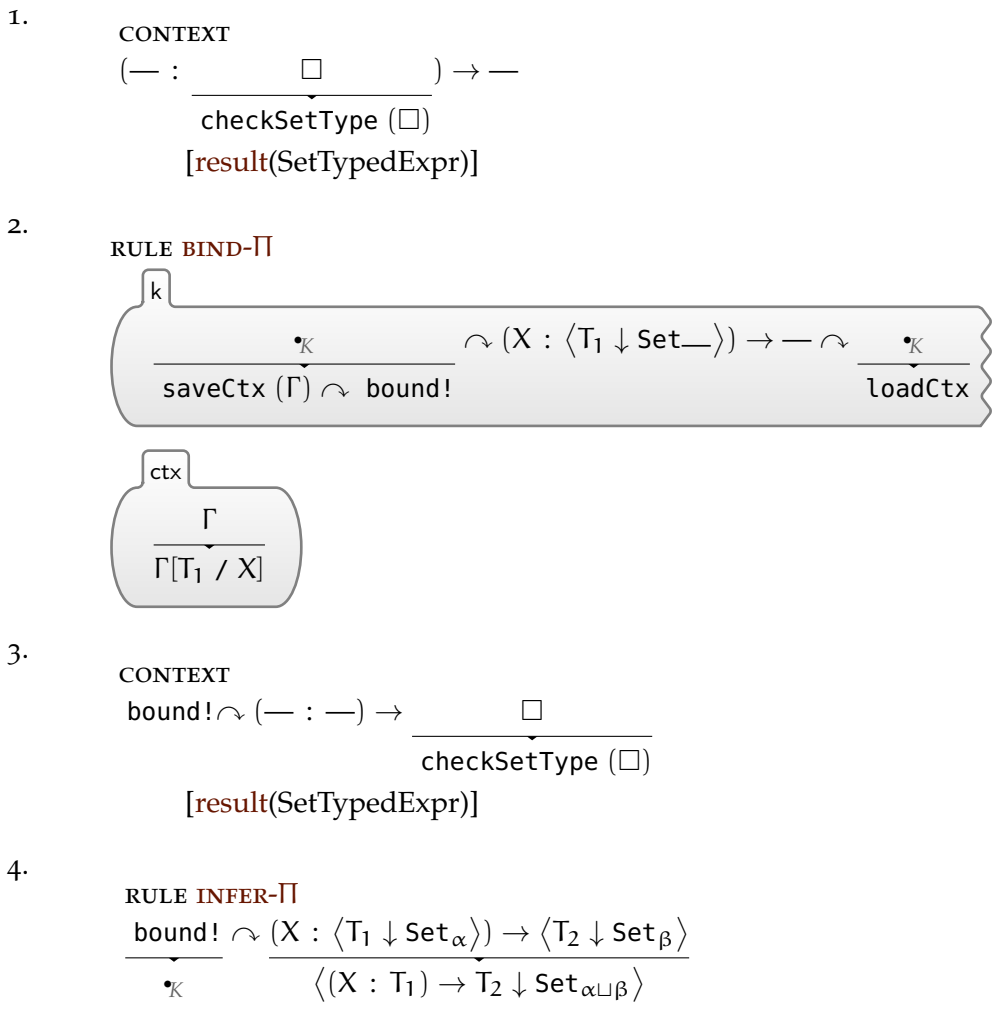


Figure 10: IKAgda implementation of type inference for Π-abstractions.

$$\frac{\Gamma \vdash e_1 \downarrow S_1 \rightsquigarrow s \qquad S_1 \to_{whnf} (x : T) \to T' \qquad \Gamma \vdash e_2 \uparrow T \rightsquigarrow t}{\Gamma \vdash e_1\ e_2 \downarrow C[t/x] \rightsquigarrow s\ t}$$

Figure 11: Type inference rule for function applications as presented by Norell [25, p. 20].

RULE INFER-APP-EXPLICIT

$$\frac{\langle E_1 \downarrow (X : T) \to T' \rangle \ \langle E_2 \downarrow T_2 \rangle}{T == T_2 \curvearrowright \texttt{unify}\ (\langle E_1\ \ E_2 \downarrow T' \rangle) \curvearrowright \texttt{tsubst}\ (X \mapsto \texttt{dedot}\ (E_2)) \curvearrowright \texttt{unifyConf} \curvearrowright \texttt{killMgu}}$$

requires $\texttt{isImplicit}(T_2) \neq_K \texttt{true} \wedge_{Bool} \texttt{canUnify}\ (T, T_2)$

Figure 12: IKAgda implementation of type inference for function applications.

for the inner type of the Π-abstraction by updating the typing context with the given type of the variable (2nd stage). To make the variable bound only within the scope of the Π-abstraction and not beyond, the auxiliary instructions `saveCtx` and `loadCtx` are used to save and restore the typing context, respectively. In addition, the blocker `bound!` is pushed at the top of `k` to prevent the binding rule to be executed more than once. The blocker is dissolved later when the type inference is finished.

The fact that we need to store the variable's type before we proceed to the body of the function space is the mark of type dependency. This would not be needed with non-dependent types because they refer only to other types. A dependent type can involve a variable whose type must hence be known in order to infer the type of the inner type.

Once bound the inner expression can be therefore checked whether it is a well-formed type as it should (3rd stage). Provided that it really is a type, the type of the whole Π-abstraction can be inferred from the as a `Set` of the greater of the two universe levels associated with the constituents of the whole expression (4th stage).

Another rule that is characteristically distinctive of dependent type systems is the inference rule for function applications. Its type-theoretic formulation is in Figure 11 while the corresponding IK rule is displayed in Figure 12.

The third premise with '↑' is a typechecking judgment. The first item of the IK rule $T{==}T_2$ adds a new unification constraint to the `mgu` cell. Obviously, there is a relationship between typechecking and unification – further explored in the next section.

If it is possibly to resolve them, the `unify` instruction applies the constraints accumulated so far. The `mgu` cell is not cleared up at that point. It is used once again to unify the contents of the configuration

(e. g., a metavariable in a type signature may get refined in this way) and just then `mgu` is emptied by the `killMgu` instruction.

The key part is `tsubst` that stands for 'type substitution'. If the variable X occurs in T', it is substituted with the *value* of the argument E_2, which is the very reason why dependent functions are called dependent. The `dedot` call is significant only when inferring the type of a left-hand side of a function clause – the same machinery is used for typing of the left-hand sides as for other function applications (see Section 4.4.2).

Hence, if the type of the argument as expected by the function and as given actually match, the application reduces to $\langle E_1\, E_2 \downarrow T' \rangle$ with all the unification constraints applied and with the variable replaced with the value of the argument if the occurred in the type of the function.

4.6.2 *Typechecking and Unification*

Typechecking = type inference + normalisation + unification

Checking of an expression E against its supposed type T consists of the following:

1. The type T' of E is inferred as described in the previous subsection.
2. The expressions T and T' are transformed into their normal forms T_{nf} and T'_{nf} respectively, cf. Section 4.5.
3. The expressions T_{nf} and T'_{nf} are attempted to be unified. If the attempt is successful, E typechecks against T. Otherwise it does not typecheck.

The problem of typechecking is thus reduced to determining if two expressions in their normal forms are unifiable. Normal forms are necessary because weak head normal forms are normalised merely along their spine (cf. Section 4.5) and in consequence can generally fail to unify in regard to those subexpressions left unreduced. Normal forms make the two expressions conform to each other as much as possible.[4]

IKAgda takes advantage of IK's built-in `unification` module for unification, with enhancements on top of it to cover certain specific issues. For instance, the normalisation needed to be done before the unification itself is not implemented in the generic module.

Moreover, a unification problem can fail to be resolved just because it involves an unnormalised pattern-matching function with some of its patterns not matched because the arguments are underspecified,

4 Recall from Section 3.3 that Agda is strongly normalising and hence the necessity of computing normal forms is not problematic. In the particular case of IKAgda, though, the statement must be qualified with remarks from Section 4.4.4.

see Section 4.5.1. The unification problem, however, can provide additional information being of consequence to the concrete form of the arguments. Namely, if the other expression that the function application ought to be unified with can be unified with one of the right-hand sides of the function then the function's arguments are instantiated in such a way that they match the respective left-hand side, resolving the overall problem. Should one of the function's arguments also contain a pattern-matching function with inconclusive matching, the algorithm is invoked recursively.

4.6.3 *Metavariables and Implicit Arguments*

𝕀KAgda internally requires a great amount of explicit type information. The types of declarations' constituents are supposed to be known, λ-abstractions need to be typed, and so forth. However, much of these data can be safely omitted in the actual Agda programs since they can be inferred automatically. To this end, metavariables are inserted in lieu of the missing terms, the position of which is either indicated by the programmer (e. g., with Agda's 'go figure' underscore), or detected and inserted automatically during typechecking.

𝕀KAgda recognises several situations in which the underscore metavariables are inserted into expressions:

1. if a type of the right-hand side of a function clause ends up to be an implicit Π-abstraction, an implicit metavariable is inserted at its end;

2. if a function is applied to an explicit argument while it expects an implicit argument, an implicit metavariable is inserted between them;

3. when the type of a function argument is an implicit Π-abstraction and the function expects another type, the argument itself gets applied to an implicit metavariable.

When an underscore is to be typed, 𝕀KAgda generates two fresh expressions: one as a metavariable to be used in place of the underscore itself and the other to serve as its type. These fresh expressions are successively refined as their surrounding expressions enforce more and more unification constraints. Ultimately, the metavariable becomes the most general expression possible to occur in the position for the whole program to be well-typed.

𝕀KAgda does not currently implement algorithm for typechecking with metavariables as given by Norell [25, sec. 3.3]. Instead, it handles metavariables using the 𝕀K-provided unification algorithm with the constraints generated during type inference.

5 CONCLUSION

In the previous chapters, I described a fragment of the the dependently typed functional programming language and interactive theorem prover Agda with emphasis on its type system. IKAgda – an executable and testable formal semantics written in IK – was presented. To my knowledge, there is no other attempt to formally describe Agda *qua* practical language so far, nor has another dependently typed language been formalised in IK.

IKAgda in its present state allows for typechecking of non-trivial Agda programs. IKAgda understands declarations of parametrised datatypes, inductively defined families, dependent functions and local definitions. It can infer metavariables and insert implicit arguments when needed. It can perform simple pattern matching in the dependently typed setting with the simultaneous instantiation of patterns that refer to datatype indices.

Still, there are many aspects in which IKAgda is lacking. From the standpoint of a IK expert, the IKAgda configuration could be restructured to use nested cells and group together data related to a single declaration instead of having them distributed across different maps as now. (Nevertheless, the advantage of the current approach is that it follows ordinary presentations of type theories which have the typing context, the function environment, etc., as maps.)

He would be perhaps interested in how IKAgda measures against the official Agda typechecker with respect to performance. The answer would not be entirely positive. In some cases IKAgda takes several minutes to process a file which is typechecked in seconds by the official system.

From the standpoint a proficient Agda user, the greatest obstacle to an immediate, practical use of IKAgda is the absence of a genuine error reporting mechanism. If a program contains a typing error, its rewriting just gets stuck at the point when the unification constraints cannot be resolved. The cause can be easily deduced in many cases but one has to be more familiar with IKAgda in order to do that swiftly.

The Agda user would also miss features that IKAgda does not support: records,[1] the module system,[2] the **with** keyword, pragmas, instance arguments, universe polymorphism, inductive-recursive definitions and coinductive datatypes. These limitations, however, could

1 They can be still introduced as standard datatypes, *sans* η-equality.
2 Module declarations are ignored, their content is processed directly.

be overcome by extending the current code without any fundamental problems.

In spite of the fact that not a few features fall beyond the scope of this thesis, it is my hope that this effort should serve as a suitable basis for any future work to the end of creating a fully complete semantics of Agda as well as an aid conducive to better understanding of dependently typed programming languages.

BIBLIOGRAPHY

[1] Agda (ver. 1) official web site, January 2009. URL http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.History. [Online; accessed 07-May-2015]. (Cited on page 11.)

[2] The Agda wiki: History, April 2014. URL http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.History. [Online; accessed 07-May-2015]. (Cited on page 11.)

[3] The Agda wiki: Reference manual, December 2014. URL http://wiki.portal.chalmers.se/agda/pmwiki.php?n=ReferenceManual.TOC. [Online; accessed 07-May-2015]. (Cited on page 21.)

[4] Andreas Abel. Towards a total dependently typed core language, September 2011. URL http://www.nii.ac.jp/shonan/seminar007/files/2011/09/talkDTP2011.pdf. Slides for a talk given at Dependently Typed Programming Workshop, Shonan Village Center, near Tokyo, Japan [Online; accessed 07-May-2015]. (Cited on page 23.)

[5] Andreas Abel. Agda: Equality, July 2012. URL http://www2.tcs.ifi.lmu.de/~abel/Equality.pdf. Lecture notes [Online; accessed 07-May-2015]. (Cited on pages 17 and 33.)

[6] Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh, and Nicolas Oury. ΠΣ: Dependent types without the sugar. In *Functional and Logic Programming*, pages 40–55. Springer, 2010. (Cited on page 23.)

[7] Lennart Augustsson. Cayenne — a language with dependent types. In *Advanced Functional Programming*, pages 240–267. Springer, 1999. (Cited on page 11.)

[8] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004. (Cited on page 11.)

[9] Denis Bogdănaş and Grigore Roşu. K-Java: A complete semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*, pages 445–456. ACM, January 2015. doi: 10.1145/2676726.2676982. (Cited on pages 5, 23, and 26.)

[10] Ana Bove and Peter Dybjer. Dependent types at work. In *Language engineering and rigorous software development*, pages 57–99. Springer, 2009. (Cited on pages 11, 16, and 18.)

[11] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda — a functional language with dependent types. In *Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009. (Cited on page 11.)

[12] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(05):552–593, 2013. (Cited on pages 11 and 22.)

[13] Thierry Coquand, Yoshiki Kinoshita, Bengt Nordström, and Makoto Takeyama. A simple type-theoretic language: Mini-TT. *From Semantics to Computer Science; Essays in Honour of Gilles Kahn*, pages 139–164, 2009. (Cited on page 23.)

[14] Chucky Ellison and Grigore Roşu. An executable formal semantics of C with applications. In *Proceedings of the 39th Symposium on Principles of Programming Languages (POPL'12)*, pages 533–544. ACM, 2012. doi: 10.1145/2103656.2103719. (Cited on page 5.)

[15] Daniele Filaretti and Sergio Maffeis. An executable formal semantics of PHP. In *ECOOP 2014–Object-Oriented Programming*, pages 567–592. Springer, 2014. (Cited on pages 5 and 21.)

[16] Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In *Algebra, Meaning, and Computation*, pages 521–540. Springer, 2006. (Cited on page 33.)

[17] Dwight Guth. A formal semantics of Python 3.3. Master's thesis, University of Illinois at Urbana-Champaign, July 2013. (Cited on pages 5 and 23.)

[18] Dorel Lucanu, Traian Florin Şerbănuţă, and Grigore Roşu. K Framework distilled. In *9th International Workshop on Rewriting Logic and its Applications*, volume 7571 of *Lecture Notes in Computer Science*, pages 31–53. Springer, 2012. Invited talk. (Cited on page 5.)

[19] Andres Löh. Programming with universes, generically, January 2012. URL `http://www.andres-loeh.de/DGP-Agda.pdf`. Slides for a talk given at the University of Tübingen [Online; accessed 07-May-2015]. (Cited on page 19.)

[20] Andres Löh, Conor McBride, and Wouter Swierstra. A tutorial implementation of a dependently typed lambda calculus. *Fundamenta informaticae*, 102(2):177–207, 2010. (Cited on page 21.)

[21] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In *Types for proofs and programs*, pages 213–237. Springer, 1994. (Cited on page 11.)

[22] Conor McBride. Faking it: Simulating dependent types in Haskell. *Journal of Functional Programming*, 12(4-5):375–392, 2002. (Cited on page 11.)

[23] José Meseguer and Grigore Roşu. The rewriting logic semantics project: A progress report. *Information and Computation*, 231:38–69, 2013. (Cited on page 23.)

[24] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory*, volume 200. Oxford University Press Oxford, 1990. (Cited on pages 11 and 16.)

[25] Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007. (Cited on pages 11, 22, 33, 35, 37, and 39.)

[26] Ulf Norell. Dependently typed programming in Agda. In *Advanced Functional Programming*, pages 230–266. Springer, 2009. (Cited on pages 11 and 13.)

[27] Nicolas Oury and Wouter Swierstra. The power of pi. In *ACM Sigplan Notices*, volume 43, pages 39–50. ACM, 2008. (Cited on page 11.)

[28] Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 428–438. ACM, June 2015. doi: 10.1145/2737924.2737991. (Cited on page 5.)

[29] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004. ISBN 0262162288. (Cited on page 34.)

[30] Grigore Roşu. K tutorial, January 2015. URL `http://www.kframework.org/index.php/K_Tutorial`. [Online; accessed 07-May-2015]. (Cited on pages 5 and 26.)

[31] Grigore Roşu and Traian Florin Şerbănuţa. K overview and SIMPLE case study. In *Proceedings of International K Workshop (K'11)*, volume 304 of *ENTCS*, pages 3–56. Elsevier, June 2014. doi: 10.1016/j.entcs.2014.05.002. (Cited on page 5.)

[32] Traian Florin Şerbănuţa, Andrei Arusoaie, David Lazar, Chucky Ellison, Dorel Lucanu, and Grigore Roşu. The K primer (version 3.3). Technical report, 2014. (Cited on page 5.)

[33] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07, pages 53–66, New York, NY, USA, 2007. ACM. ISBN 1-59593-393-X. doi: 10.1145/1190315.1190324. (Cited on page 22.)

[34] Simon Thompson. *Type Theory and Functional Programming.* International computer science series. Addison-Wesley, 1991. ISBN 978-0-201-41667-1. (Cited on page 11.)

A

QUICKSTART GUIDE

The versions of software used during the development of IKAgda are:

- IK Framework, version 3.4 (released 15-Oct-2014);
- the official Agda distribution, version 2.4.2.2 (released 26-Nov-2014).

IKAgda should be best set up with these versions. The next section explains how these should be installed with the functionality of IKAgda itself demonstrated subsequently.

The actual IKAgda code and other necessary files are to be found in the archive provided together with this or in the Git repository living at `https://github.com/andrejtokarcik/agda-semantics`. The structure of the project repository is self-explanatory.

A.1 INSTALLATION OF THE DEPENDENCIES

IK Framework can be downloaded in the form of precompiled binaries from `http://www.kframework.org/index.php/K_tool_binaries`.

Agda, however, must be compiled from source because it serves as the parser and scope analyser for IKAgda, which requires some changes to be made within the code of the official typechecker as per Section 4.2.1. The relevant Agda archive can be obtained at `http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.Download`.

After the archive is downloaded and extracted, the patch `agda-2.4.2.2-kshow.diff` from the IKAgda package should be applied.

You can patch the Agda sources by going into the clean `agda-2.4.2.2` directory and running this command in shell:

```
$ cd [path to the agda-2.4.2.2 directory]
$ patch -p1 < [path to agda-2.4.2.2-kshow.diff]
# the output should be:
patching file Agda.cabal
patching file src/full/Agda/Interaction/Imports.hs
patching file src/full/Agda/KShow.hs
patching file src/full/Agda/Syntax/Concrete/KShow.hs
patching file src/full/Agda/Syntax/Concrete/Name.hs
patching file src/full/Agda/Syntax/Concrete/Pretty.hs
patching file src/full/Agda/Syntax/Concrete.hs
patching file src/full/Agda/Syntax/Translation/
    AbstractToConcrete.hs
```

The code then needs to be built as it would normally be, according to the `README` file from the Agda archive.[1] Afterwards copy the resulting binary located at `dist/build/agda/agda` so that it is present in the shell's path as `agda-kshow`. IKAgda's scripts expect the external parser to exist under this name specifically.

Note that the patched version of Agda cannot be directly employed in its capacity as typechecker – its execution flow is stopped prematurely to return the AST for IKAgda. The patched version should be therefore installed in parallel with a vanilla distribution of the Agda system.

A.2 WORKING WITH THE SEMANTICS

Once both IK and Agda are prepared, IKAgda can be also given a few CPU cycles. Go to the directory with the semantics files. It should contain a file named `agda.k` which is the entry point to the whole language definition. First we need to compile it for which the `kompile.sh` script bundled with IKAgda comes in handy. You could use IK's `kompile` executable directly but the script wraps around `kompile` executable in order to supply some useful options. Keep in mind, though, that the process can eat up two gigabytes of memory, or even more.

```
$ cd [path to the agda-semantics directory]
$ ./kompile.sh
# reports info about files about stages of the compilation,
    concluding with:
Number of Modules                          =   13
Number of Sentences                        =  350
Number of Productions                      =  207
Number of Cells                            =    8
```

The `kompile` process produces the `agda-kompiled` subdirectory. With the definition compiled it is now possible to let some real Agda programs typecheck (or not) with IKAgda.

The script `krun.sh` instructs the standard `krun` tool to use the manually compiled Agda system for parsing (with scope analysis done as a 'side effect'). It also saves the `krun` output (which tends to be pretty large) to the `out` file in the current directory while still printing to the console.

Take the literate Agda file of Section 3.1 that is included in the test suite for example. The `krun` output is quite unordered and verbose and so the data are regrouped to aid comprehension. Moreover, only those parts pertaining to the vector lookup function are displayed:

```
$ ./krun.sh ./tests/DepStructs.lagda
# ...
<k>
```

1 Long story short, make sure you have all the software prerequisites installed and then run `cabal install` in the patched directory.

```
   .K
</k>
<mgu>
   subst(.KList)
</mgu>
<ctx-stack>
   .List
</ctx-stack>
<sig>
   Data ( "8469" ) |-> Set ( 0 ) # the number is a code for ℕ
   Con ( "zero" ) |-> Data ( "8469" )
   Con ( "succ" ) |-> ( #symVariable(30) : Data ( "8469" ) ) ->
       Data ( "8469" )

   Data ( "Fin" ) |-> ( #symVariable(466) : Data ( "8469" ) ) ->
        Set ( 0 )
   Con ( "fzero" ) |-> { #symVariable(475) : Data ( "8469" ) }
       -> (Data ( "Fin" ) (Con ( "succ" ) #symVariable(475)))
   Con ( "fsucc" ) |-> { #symVariable(489) : Data ( "8469" ) }
       -> ( #symVariable(488) : (Data ( "Fin" ) #symVariable
       (489)) ) -> (Data ( "Fin" ) (Con ( "succ" ) #symVariable
       (489)))

   Data ( "Vec" ) |-> ( #symVariable(432) : Set ( 0 ) ) -> (
       #symVariable(431) : Data ( "8469" ) ) -> Set ( 0 )
   Con ( "<>" ) |-> { #symVariable(441) : Set ( 0 ) } -> ((Data
       ( "Vec" ) #symVariable(441)) Con ( "zero" ))
   Con ( "_,_" ) |-> { #symVariable(465) : Set ( 0 ) } -> {
       #symVariable(464) : Data ( "8469" ) } -> ( #symVariable
       (462) : #symVariable(465) ) -> ( #symVariable(463) : ((
       Data ( "Vec" ) #symVariable(465)) #symVariable(464)) ) ->
        ((Data ( "Vec" ) #symVariable(465)) (Con ( "succ" )
       #symVariable(464)))

   Fun ( "_[_]" ) |-> { #symVariable(505) : Set ( 0 ) } -> {
       #symVariable(504) : Data ( "8469" ) } -> ( #symVariable
       (502) : ((Data ( "Vec" ) #symVariable(505)) #symVariable
       (504)) ) -> ( #symVariable(503) : (Data ( "Fin" )
       #symVariable(504)) ) -> #symVariable(505)

   # ...
</sig>
<ctx>
   .Map
</ctx>
<meta>
   .Map
</meta>
<ncat>
   Name ( "8469" ) |-> Data ( "8469" )
   Name ( "zero" ) |-> Con ( "zero" )
   Name ( "succ" ) |-> Con ( "succ" )
```

```
    Name ( "Fin" ) |-> Data ( "Fin" )
    Name ( "fzero" ) |-> Con ( "fzero" )
    Name ( "fsucc" ) |-> Con ( "fsucc" )
    Name ( "Vec" ) |-> Data ( "Vec" )
    Name ( "<>" ) |-> Con ( "<>" )
    Name ( "_,_" ) |-> Con ( "_,_" )
    Name ( "_[_]" ) |-> Fun ( "_[_]" )
    # ...
</ncat>
<fun>
    Fun ( "_[_]" ) |-> 'fc(
        ((((#symFunClauses(647) { (DotP ( #symVariable(643) )) })
            { (Con ( "succ" ) DotP ( #symExpr(635) )) }) ((((Con (
             "_,_" ) { #symVariable(643) }) { (DotP ( #symExpr
            (635) )) }) Var ( "x" )) #symExpr(539))) (Con ( "fzero
            " ) { #symExpr(635) })) = Var ( "x" ),,
        (((( #symFunClauses(793) { (DotP ( #symVariable(789) )) })
             { (Con ( "succ" ) 'DotP ( #symExpr(747) )) }) ((((Con
             ( "_,_" ) { #symVariable(789) }) { ( DotP ( #symExpr
            (747) )) }) #symExpr(658)) Var ( "xs" ))) ((Con ( "
            fsucc" ) { #symExpr(747) }) Var ( "i" ))) = ((((Fun (
            "_[_]" ) { #symVariable(789) }) { #symExpr(747) }) Var
             ( "xs" )) Var ( "i" )))
    # ...
</fun>
```

There are obviously some discrepancies between the presentation in the rest of the thesis and this pile of characters. These should be cleared up immediately. The IKAgda configuration cells of Section 4.3.1 are here displayed as XML tags in ASCII format. The '$\mapsto$' symbol used in maps is written as '`|->`'. The '`#sym`' things were created as fresh variables of the corresponding syntactic sort. Also, since IK is no friend with unicode (in total contrast to Agda), a numeric code is used instead of '$\mathbb{N}$' as the identifier for the type of natural numbers. To be able to decipher IKAgda outputs in general may need some additional practice but these remarks should be sufficient to get you started.

The contents of the `k` cell is '`.K`', which is the ASCII notation for $\bullet_K$, which in turn implies that the input program has been completely consumed. The fact that the `k` cell contains no leftovers indicates that IKAgda considers the program to be well-typed. The `ctx`, `meta` and `mgu` cells have been also cleared up, as expected of a successful run. The type signatures gathered during the typechecking are stored in the `sig` cell for examination, whereas the function bodies are in the `fun` cell. For further details on how these are determined and represented refer to Section 4.4.

Finally, the semantics comes with a set of Agda programs under the `tests` directory many of which were collected from various Agda tutorials. The set works as a test suite that can be executed simply with the command:

```
$ ./ktest.sh
```

The script runs the test suite and for each test input compares its present IKAgda output with the expected result. If all the test cases pass, the script finishes with a green `SUCCESS` printed to the console.

The `tests` directory is divided into `covered` and `beyond`. The former subdirectory contains programs that use only those Agda features that are implemented in IKAgda whereas the latter contains Agda files that fall beyond the scope of IKAgda in one way or another. One can examine the cases in `beyond` to get an idea about the limitations of the IKAgda typechecker. Only the `covered` Agda programs are executed by the `ktest` script.

COLOPHON

This document was typeset using LaTeXwith the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography *The Elements of Typographic Style*. `classicthesis` is available:

`http://code.google.com/p/classicthesis/`

Final Version as of 25th May 2015 (`classicthesis` version 4.1).