Dependent Type Theory with Parameterized First-Order Data Types and Well-Founded Recursion – Opponent's Presentation of David Wahlstedt's PhD Thesis

Ralph Matthes

Institut de Recherche en Informatique de Toulouse (IRIT), CNRS
Équipe ACADIE
(Assistance à la Certification de Systèmes Distribués et Embarqués)

Department of Computer Science and Engineering
Chalmers University of Technology, Göteborg, Sweden
September 14, 2007

Abstract

This is the presentation of David Wahlstedt's PhD thesis, from the point of view of the Faculty Opponent.

Outline

Outline

Outline

Recall the title:

Dependent Type Theory with Parameterized
First-Order Data Types and Well-Founded Recursion

- Theory
- Type Theory
- Dependent Type Theory
- Data Types
- First-Order Data Types
- Parameterized First-Order Data Types
- Recursion
- Wellfounded Recursion

Recall the title:

Dependent Type Theory with Parameterized
First-Order Data Types and Well-Founded Recursion

- Theory
- Type Theory
- Dependent Type Theory
- Data Types
- First-Order Data Types
- Parameterized First-Order Data Types
- Recursion
- Wellfounded Recursion

Theory

This is a thesis in theoretical computer science:

- precise mathematical language
- definitions are not the end product but the raw material
- theorems play an important role
- theorems come with elaborate proofs
- examples do not abound but are an important means of justifying the whole endeavour
- this thesis: no prototypical implementation

Theory

This is a thesis in theoretical computer science:

- precise mathematical language
- definitions are not the end product but the raw material
- theorems play an important role
- theorems come with elaborate proofs
- examples do not abound but are an important means of justifying the whole endeavour
- this thesis: no prototypical implementation

Theory

This is a thesis in theoretical computer science:

- precise mathematical language
- definitions are not the end product but the raw material
- theorems play an important role
- theorems come with elaborate proofs
- examples do not abound but are an important means of justifying the whole endeavour
- this thesis: no prototypical implementation

Type Theory

This thesis contributes to Type Theory:

- formal languages where expressions are being typed
- assignment of types to raw expressions is a non-trivial activity
- these formal languages have logical and more operational readings
- the lambda-calculus underlies most of those formalisms
- therefore, these are (idealized) functional programming languages
- having a type may induce important general (meta-theoretical) consequences:
  - termination
  - reducts have the same type (= subject reduction)
- typing may have important general (meta-theoretical) properties such as decidability

Type Theory

This thesis contributes to Type Theory:

- formal languages where expressions are being typed
- assignment of types to raw expressions is a non-trivial activity
- these formal languages have logical and more operational readings
- the lambda-calculus underlies most of those formalisms
- therefore, these are (idealized) functional programming languages
- having a type may induce important general (meta-theoretical) consequences:
  - termination
  - reducts have the same type (= subject reduction)
- typing may have important general (meta-theoretical) properties such as decidability

Type Theory

This thesis contributes to Type Theory:

- formal languages where expressions are being typed
- assignment of types to raw expressions is a non-trivial activity
- these formal languages have logical and more operational readings
- the lambda-calculus underlies most of those formalisms
- therefore, these are (idealized) functional programming languages
- having a type may induce important general (meta-theoretical) consequences:
  - termination
  - reducts have the same type (= subject reduction)
- typing may have important general (meta-theoretical) properties such as decidability

Type Theory

This thesis contributes to Type Theory:

- formal languages where expressions are being typed
- assignment of types to raw expressions is a non-trivial activity
- these formal languages have logical and more operational readings
- the lambda-calculus underlies most of those formalisms
- therefore, these are (idealized) functional programming languages
- having a type may induce important general (meta-theoretical) consequences:
  - termination
  - reducts have the same type ($=$ subject reduction)
- typing may have important general (meta-theoretical) properties such as decidability

Dependent Type Theory

Types may depend on inhabitants (“objects”) of types.

This occurs only too naturally with predicate logic if formulae are seen as types: The statement “n is odd” is a type that depends on the natural number n.

Not as common with operational reading / in programming languages, but with standard example: the type of lists of precisely n elements taken from a given type A (“vectors”). The A is “just a parameter”, but the n makes it a dependent type.

With a more uniform view, A is an object in a suitable “kind”, here the kind of all “small” types. If all dependencies are as simple as that, then we do not need the specific methods of dependent type theory that show up throughout this thesis.

Dependent Type Theory

Types may depend on inhabitants ("objects") of types.

This occurs only too naturally with predicate logic if formulae are seen as types: The statement "n is odd" is a type that depends on the natural number n.

Not as common with operational reading / in programming languages, but with standard example: the type of lists of precisely n elements taken from a given type A ("vectors"). The A is "just a parameter", but the n makes it a dependent type.

With a more uniform view, A is an object in a suitable "kind", here the kind of all "small" types. If all dependencies are as simple as that, then we do not need the specific methods of dependent type theory that show up throughout this thesis.

Dependent Type Theory

Types may depend on inhabitants ("objects") of types.

This occurs only too naturally with predicate logic if formulae are seen as types: The statement "*n* is odd" is a type that depends on the natural number *n*.

Not as common with operational reading / in programming languages, but with standard example: the type of lists of precisely *n* elements taken from a given type *A* ("vectors"). The *A* is "just a parameter", but the *n* makes it a dependent type.

With a more uniform view, *A* is an object in a suitable "kind", here the kind of all "small" types. If all dependencies are as simple as that, then we do not need the specific methods of dependent type theory that show up throughout this thesis.

Data Types

Data types are sets whose elements are formed by following certain construction rules. They are formalized by element constructors such as *nil* and *cons* for lists.

In this thesis, all elements are well-founded, hence are not built by infinitely piling up constructors. However, arguments of constructors may be functions (also into the data type, for infinitely branching trees), hence elements of data types may be infinite in nature (although finitely described by a lambda term).

Operations on these data types are at the heart of this thesis: "defined constants". They may assign objects in data types but also types themselves.

This is not algebraic specification in the equational sense, but constant definitions come in the form of rewrite rules. Thus, it is higher-order rewriting in addition to β.

Data Types

Data types are sets whose elements are formed by following certain construction rules. They are formalized by element constructors such as *nil* and *cons* for lists.

In this thesis, all elements are well-founded, hence are not built by infinitely piling up constructors. However, arguments of constructors may be functions (also into the data type, for infinitely branching trees), hence elements of data types may be infinite in nature (although finitely described by a lambda term).

Operations on these data types are at the heart of this thesis: "defined constants". They may assign objects in data types but also types themselves.

This is not algebraic specification in the equational sense, but constant definitions come in the form of rewrite rules. Thus, it is higher-order rewriting in addition to β.

Data Types

Data types are sets whose elements are formed by following certain construction rules. They are formalized by element constructors such as *nil* and *cons* for lists.

In this thesis, all elements are well-founded, hence are not built by infinitely piling up constructors. However, arguments of constructors may be functions (also into the data type, for infinitely branching trees), hence elements of data types may be infinite in nature (although finitely described by a lambda term).

Operations on these data types are at the heart of this thesis: "defined constants". They may assign objects in data types but also types themselves.

This is not algebraic specification in the equational sense, but constant definitions come in the form of rewrite rules. Thus, it is higher-order rewriting in addition to β.

Data Types

Data types are sets whose elements are formed by following certain construction rules. They are formalized by element constructors such as *nil* and *cons* for lists.

In this thesis, all elements are well-founded, hence are not built by infinitely piling up constructors. However, arguments of constructors may be functions (also into the data type, for infinitely branching trees), hence elements of data types may be infinite in nature (although finitely described by a lambda term).

Operations on these data types are at the heart of this thesis: "defined constants". They may assign objects in data types but also types themselves.

This is not algebraic specification in the equational sense, but constant definitions come in the form of rewrite rules. Thus, it is higher-order rewriting in addition to β.

First-Order Data Types

This thesis is not just about adding pure algebraic types to a rich framework.

The universe *Set* of data types already comes with a built-in dependent function space: The constant Π of type

$$(x : Set, El\,x \rightarrow Set) \rightarrow Set,$$

that is the paradigmatic higher-order data type constructor.

But the "user" can only supply new data types by means of a set constructor of a type $Set^n \rightarrow Set$, i. e., $Set \rightarrow Set \rightarrow \ldots \rightarrow Set$ with $\rightarrow$ associating to the right. $Set^n \rightarrow Set$ is a first-order signature on the framework level.

First-Order Data Types

This thesis is not just about adding pure algebraic types to a rich framework.

The universe *Set* of data types already comes with a built-in dependent function space: The constant Π of type

$$(x : Set, El\, x \to Set) \to Set,$$

that is the paradigmatic higher-order data type constructor.

But the “user” can only supply new data types by means of a set constructor of a type $Set^n \to Set$, i. e., $Set \to Set \to \ldots \to Set$ with $\to$ associating to the right. $Set^n \to Set$ is a first-order signature on the framework level.

First-Order Data Types

This thesis is not just about adding pure algebraic types to a rich framework.

The universe *Set* of data types already comes with a built-in dependent function space: The constant Π of type

$$(x : Set, El\, x \to Set) \to Set,$$

that is the paradigmatic higher-order data type constructor.

But the “user” can only supply new data types by means of a set constructor of a type $Set^n \to Set$, i. e., $Set \to Set \to \ldots \to Set$ with $\to$ associating to the right. $Set^n \to Set$ is a first-order signature on the framework level.

Parameterized First-Order Data Types

Why $Set^n \to Set$ and not just Set? Because this allows n set parameters to be given to the data type.

The most standard example: $List : Set \to Set$, where the parameter is the type from which the elements are taken.

The parameter may vary between input type and output type of element constructors, thus covering also "nested data types" (see the example on page 43 with powerlists).

Parameterized First-Order Data Types

Why $\mathit{Set}^n \to \mathit{Set}$ and not just Set? Because this allows n set parameters to be given to the data type.

The most standard example: $\mathit{List} : \mathit{Set} \to \mathit{Set}$, where the parameter is the type from which the elements are taken.

The parameter may vary between input type and output type of element constructors, thus covering also "nested data types" (see the example on page 43 with powerlists).

Recursion

The defined constants are ruled by descriptions how to calculate the result from the arguments. One has several descriptions, corresponding to the different element constructors that form the arguments if they belong to user-defined data types.
For the factorial function, one would set

$$\mathit{fact}\,0 \to s\,0, \qquad \mathit{fact}(s\,n) \to \mathit{mult}\,(s\,n)\,(\mathit{fact}\,n).$$

This uses two important concepts:

- pattern matching
- recursion

If *fact* gets an argument with a leading *s*, it chops off that *s* and binds the remaining expression to the formal parameter *n*, and then evaluates the expression to the right-hand side.

Recursion

The defined constants are ruled by descriptions how to calculate the result from the arguments. One has several descriptions, corresponding to the different element constructors that form the arguments if they belong to user-defined data types.
For the factorial function, one would set

$$\mathit{fact}\, 0 \to s\, 0, \qquad \mathit{fact}(s\, n) \to \mathit{mult}\,(s\, n)\,(\mathit{fact}\, n).$$

This uses two important concepts:

- pattern matching
- recursion

If *fact* gets an argument with a leading s, it chops off that s and binds the remaining expression to the formal parameter n, and then evaluates the expression to the right-hand side.

Wellfounded Recursion

Recursion is when the defined constant also appears on the right-hand side.

A major question is that of termination of the execution of recursive programs. In other words: Is the recursion wellfounded?

A binary relation $>$ is called wellfounded if there is no infinite sequence $a_1 > a_2 > a_3 > \ldots$ For the execution of a program, one can be satisfied if no infinite sequence of recursive calls is stringed together, hence if the call relation is wellfounded.

In the end, one just wants that a good implementation of the reduction process (executing the framework rules and the user-defined rules) terminates on all inputs. Hence, the reduction relation does not need to be wellfounded but just weakly terminating to some normal form, and the implementor has to provide a strategy that attains this normal form.

Wellfounded Recursion

Recursion is when the defined constant also appears on the right-hand side.

A major question is that of termination of the execution of recursive programs. In other words: Is the recursion wellfounded?

A binary relation $>$ is called wellfounded if there is no infinite sequence $a_1 > a_2 > a_3 > \ldots$ For the execution of a program, one can be satisfied if no infinite sequence of recursive calls is stringed together, hence if the call relation is wellfounded.

In the end, one just wants that a good implementation of the reduction process (executing the framework rules and the user-defined rules) terminates on all inputs. Hence, the reduction relation does not need to be wellfounded but just weakly terminating to some normal form, and the implementor has to provide a strategy that attains this normal form.

Wellfounded Recursion

Recursion is when the defined constant also appears on the right-hand side.

A major question is that of termination of the execution of recursive programs. In other words: Is the recursion wellfounded?

A binary relation $>$ is called wellfounded if there is no infinite sequence $a_1 > a_2 > a_3 > \ldots$ For the execution of a program, one can be satisfied if no infinite sequence of recursive calls is stringed together, hence if the call relation is wellfounded.

In the end, one just wants that a good implementation of the reduction process (executing the framework rules and the user-defined rules) terminates on all inputs. Hence, the reduction relation does not need to be wellfounded but just weakly terminating to some normal form, and the implementor has to provide a strategy that attains this normal form.

Wellfounded Recursion

Recursion is when the defined constant also appears on the right-hand side.

A major question is that of termination of the execution of recursive programs. In other words: Is the recursion wellfounded?

A binary relation $>$ is called wellfounded if there is no infinite sequence $a_1 > a_2 > a_3 > \ldots$ For the execution of a program, one can be satisfied if no infinite sequence of recursive calls is stringed together, hence if the call relation is wellfounded.

In the end, one just wants that a good implementation of the reduction process (executing the framework rules and the user-defined rules) terminates on all inputs. Hence, the reduction relation does not need to be wellfounded but just weakly terminating to some normal form, and the implementor has to provide a strategy that attains this normal form.

Outline

Overall Goal, Generally

Programming of and reasoning about technical reality in a mathematically precise way. For this, we need

- programming languages
- logics
- program logics

Programming languages may be too liberal so that safety-critical applications become hazardous - even with a lot of testing.
Run-time type systems ensure that senseless commands are not executed.
Static analysis allows to exclude run-time type errors already by inspection of the source code. Program logics often ensure partial correctness: computed results are correct, but such specifications have to be validated individually against the "requirements baseline".

Overall Goal, Generally

Programming of and reasoning about technical reality in a mathematically precise way. For this, we need

- programming languages
- logics
- program logics

Programming languages may be too liberal so that safety-critical applications become hazardous - even with a lot of testing. Run-time type systems ensure that senseless commands are not executed.

Static analysis allows to exclude run-time type errors already by inspection of the source code. Program logics often ensure partial correctness: computed results are correct, but such specifications have to be validated individually against the "requirements baseline".

Overall Goal, Generally

Programming of and reasoning about technical reality in a mathematically precise way. For this, we need

- programming languages
- logics
- program logics

Programming languages may be too liberal so that safety-critical applications become hazardous - even with a lot of testing.
Run-time type systems ensure that senseless commands are not executed.
Static analysis allows to exclude run-time type errors already by inspection of the source code. Program logics often ensure partial correctness: computed results are correct, but such specifications have to be validated individually against the “requirements baseline”.

Overall Goal, More Specifically

In type theory, totality is a major issue that does not depend on the specific program under investigation: it is the “non-functional requirement” that results are obtained in finite time (mostly not taking into account time and space consumption quantitatively).

This thesis is about guaranteeing that the process of repeated simplification of all typable expressions of the language is finite. And simplification is done by replacing formal parameters with actual arguments (β-reduction) and by unfolding user-defined recursive definitions of functions in terms and in types.

In a dependently typed system, this has important consequences on the type-checking itself (decidability). It also provides logical consistency.

Overall Goal, More Specifically

In type theory, totality is a major issue that does not depend on the specific program under investigation: it is the "non-functional requirement" that results are obtained in finite time (mostly not taking into account time and space consumption quantitatively).

This thesis is about guaranteeing that the process of repeated simplification of all typable expressions of the language is finite. And simplification is done by replacing formal parameters with actual arguments (β-reduction) and by unfolding user-defined recursive definitions of functions in terms and in types.

In a dependently typed system, this has important consequences on the type-checking itself (decidability). It also provides logical consistency.

Overall Goal, More Specifically

In type theory, totality is a major issue that does not depend on the specific program under investigation: it is the “non-functional requirement” that results are obtained in finite time (mostly not taking into account time and space consumption quantitatively).

This thesis is about guaranteeing that the process of repeated simplification of all typable expressions of the language is finite. And simplification is done by replacing formal parameters with actual arguments (β-reduction) and by unfolding user-defined recursive definitions of functions in terms and in types.

In a dependently typed system, this has important consequences on the type-checking itself (decidability). It also provides logical consistency.

Overall Goal, This Thesis

- A precise definition of a variant of Martin-Löf's type theory that does not come with standard recursors but allows an important freedom in specifying defined constants, using pattern matching.
- Proving the important meta-theoretic properties of confluence and subject reduction, by factoring out results for pattern matching.
- Characterizing the typing of normal forms in an implementation-oriented manner.
- Giving a semantics of the language that allows to deduce normalization if all defined constants are reducible (a reducible signature).

Overall Goal, This Thesis

- A precise definition of a variant of Martin-Löf's type theory that does not come with standard recursors but allows an important freedom in specifying defined constants, using pattern matching.
- Proving the important meta-theoretic properties of confluence and subject reduction, by factoring out results for pattern matching.
- Characterizing the typing of normal forms in an implementation-oriented manner.
- Giving a semantics of the language that allows to deduce normalization if all defined constants are reducible (a reducible signature).

Overall Goal, This Thesis

- A precise definition of a variant of Martin-Löf's type theory that does not come with standard recursors but allows an important freedom in specifying defined constants, using pattern matching.
- Proving the important meta-theoretic properties of confluence and subject reduction, by factoring out results for pattern matching.
- Characterizing the typing of normal forms in an implementation-oriented manner.
- Giving a semantics of the language that allows to deduce normalization if all defined constants are reducible (a reducible signature).

Overall Goal, This Thesis

- A precise definition of a variant of Martin-Löf's type theory that does not come with standard recursors but allows an important freedom in specifying defined constants, using pattern matching.
- Proving the important meta-theoretic properties of confluence and subject reduction, by factoring out results for pattern matching.
- Characterizing the typing of normal forms in an implementation-oriented manner.
- Giving a semantics of the language that allows to deduce normalization if all defined constants are reducible (a reducible signature).

Overall Goal, This Thesis, Cont'd

- Showing that all defined constants are reducible if the call relation is well-founded.
- Showing that the call relation is well-founded in the case of size-change termination.
- Showing decidability of type checking for normal expressions and for patterns and logical consistency, given a reducible signature.
- Giving a decision procedure for a stepwise extension of the signature, keeping reducibility.

Overall Goal, This Thesis, Cont'd

- Showing that all defined constants are reducible if the call relation is well-founded.
- Showing that the call relation is well-founded in the case of size-change termination.
- Showing decidability of type checking for normal expressions and for patterns and logical consistency, given a reducible signature.
- Giving a decision procedure for a stepwise extension of the signature, keeping reducibility.

Overall Goal, This Thesis, Cont'd

- Showing that all defined constants are reducible if the call relation is well-founded.
- Showing that the call relation is well-founded in the case of size-change termination.
- Showing decidability of type checking for normal expressions and for patterns and logical consistency, given a reducible signature.
- Giving a decision procedure for a stepwise extension of the signature, keeping reducibility.

Important Other Approaches

- Defining what is a smaller expression by using higher-order versions of the recursive path ordering.
- The analysis of recursive calls by means of ensuring that it is done with arguments that are smaller does no longer work for non-strictly positive fixed points. Even when structural descent is possible, also the following are studied:
- Recursion operators
- Impredicative encodings
- Recursion principles in the style of Mendler (1987) that use universal quantification over types (this was the first system with type-based termination)
- Sized types (Lars Pareto 2000, . . .): data types with size annotations

Important Other Approaches

- Defining what is a smaller expression by using higher-order versions of the recursive path ordering.
- The analysis of recursive calls by means of ensuring that it is done with arguments that are smaller does no longer work for non-strictly positive fixed points. Even when structural descent is possible, also the following are studied:
- Recursion operators
- Impredicative encodings
- Recursion principles in the style of Mendler (1987) that use universal quantification over types (this was the first system with type-based termination)
- Sized types (Lars Pareto 2000, . . .): data types with size annotations

Important Other Approaches

- Defining what is a smaller expression by using higher-order versions of the recursive path ordering.
- The analysis of recursive calls by means of ensuring that it is done with arguments that are smaller does no longer work for non-strictly positive fixed points. Even when structural descent is possible, also the following are studied:
- Recursion operators
- Impredicative encodings
- Recursion principles in the style of Mendler (1987) that use universal quantification over types (this was the first system with type-based termination)
- Sized types (Lars Pareto 2000, . . .): data types with size annotations

Outline

Read the whole thesis. It is just 120 pages long and very well written.

- An introduction of 24 pages: Everything one wants to know before really reading the technical part. And two illuminating discussions about non-trivial examples: Berry's majority function (p. 16) and Vogel's trick (pp. 21, 22).
- The "syntax" chapter of 39 pages with the system definition, the confluence proof (following the usual method), the pattern-matching facility through atomic neighbourhood and compound neighbourhood, some examples and the usual (and technically still demanding) apparatus of meta-theory leading through generation (for the system in Curry-style formulation!) to subject reduction. The neighbourhoods play an important role here. An implementation-oriented typing system for β-normal forms with soundness and completeness proof (thus a characterization of the restricted typing relation).

Read the whole thesis. It is just 120 pages long and very well written.

- An introduction of 24 pages: Everything one wants to know before really reading the technical part. And two illuminating discussions about non-trivial examples: Berry's majority function (p. 16) and Vogel's trick (pp. 21, 22).
- The "syntax" chapter of 39 pages with the system definition, the confluence proof (following the usual method), the pattern-matching facility through atomic neighbourhood and compound neighbourhood, some examples and the usual (and technically still demanding) apparatus of meta-theory leading through generation (for the system in Curry-style formulation!) to subject reduction. The neighbourhoods play an important role here. An implementation-oriented typing system for β-normal forms with soundness and completeness proof (thus a characterization of the restricted typing relation).

Read the whole thesis. It is just 120 pages long and very well written.

- An introduction of 24 pages: Everything one wants to know before really reading the technical part. And two illuminating discussions about non-trivial examples: Berry's majority function (p. 16) and Vogel's trick (pp. 21, 22).
- The "syntax" chapter of 39 pages with the system definition, the confluence proof (following the usual method), the pattern-matching facility through atomic neighbourhood and compound neighbourhood, some examples and the usual (and technically still demanding) apparatus of meta-theory leading through generation (for the system in Curry-style formulation!) to subject reduction. The neighbourhoods play an important role here. An implementation-oriented typing system for β-normal forms with soundness and completeness proof (thus a characterization of the restricted typing relation).

The second last 20%

- The "semantics" chapter of 20 pages. This is reducibility semantics, a. k. a. "computability", a strengthening (cf. Prop. 3.3.3) of weak normalization that is closed under well-typed application by definition. The reducibility predicate is a one-place variant of a logical relation. It specifies which types are reducible and which terms are reducible w. r. t. a reducible type. This is done by a simultaneous inductive-recursive definition, called "specification", since its set-theoretic justification is only sketched (pp. 70–73).
    - usual soundness theorem (Lemma 3.5.1): typable implies reducible (under substitution) if defined constants are reducible
    - call relation formally defined
    - Key lemma: call relation wellfounded and types of the defined constants reducible imply defined constants reducible

The second last 20%

- The "semantics" chapter of 20 pages. This is reducibility semantics, a. k. a. "computability", a strengthening (cf. Prop. 3.3.3) of weak normalization that is closed under well-typed application by definition. The reducibility predicate is a one-place variant of a logical relation. It specifies which types are reducible and which terms are reducible w. r. t. a reducible type. This is done by a simultaneous inductive-recursive definition, called "specification", since its set-theoretic justification is only sketched (pp. 70–73).
  - usual soundness theorem (Lemma 3.5.1): typable implies reducible (under substitution) if defined constants are reducible
  - call relation formally defined
  - Key lemma: call relation wellfounded and types of the defined constants reducible imply defined constants reducible

The last 20%: 3 chapters

- Chapter 4 (5 pages) explains the size-change principle and shows that it implies wellfoundedness of the call relation
- Chapter 5 (12 pages) puts everything together to obtain decidability results for the type-checking problems mentioned earlier and a decision procedure for a stratification of the types and rules of the user-defined constants. Finally, logical consistency given exhaustiveness of the pattern-matching definitions.
- Chapter 6 (4 pages) is the final discussion:
    - What has been obtained (in clear words)
    - What has been obtained since the Licentiate Thesis
    - The modularity aspect of the whole development is stressed
    - Future work that could follow naturally

The last 20%: 3 chapters

- Chapter 4 (5 pages) explains the size-change principle and shows that it implies wellfoundedness of the call relation
- Chapter 5 (12 pages) puts everything together to obtain decidability results for the type-checking problems mentioned earlier and a decision procedure for a stratification of the types and rules of the user-defined constants. Finally, logical consistency given exhaustiveness of the pattern-matching definitions.
- Chapter 6 (4 pages) is the final discussion:
  - What has been obtained (in clear words)
  - What has been obtained since the Licentiate Thesis
  - The modularity aspect of the whole development is stressed
  - Future work that could follow naturally

The last 20%: 3 chapters

- Chapter 4 (5 pages) explains the size-change principle and shows that it implies wellfoundedness of the call relation
- Chapter 5 (12 pages) puts everything together to obtain decidability results for the type-checking problems mentioned earlier and a decision procedure for a stratification of the types and rules of the user-defined constants. Finally, logical consistency given exhaustiveness of the pattern-matching definitions.
- Chapter 6 (4 pages) is the final discussion:
  - What has been obtained (in clear words)
  - What has been obtained since the Licentiate Thesis
  - The modularity aspect of the whole development is stressed
  - Future work that could follow naturally

Outline

Rough Sketch of the System

The types are *Set* (the universe of data types), the type *El* t of elements of a type code, represented by a term t of type *Set*, and the dependent function type (of the framework).

Elements of *Set* come from the Π (on the level of data types) and from the user-defined parameterized data types d. Elements of Π types are constructed by help of a primitive constant *fun*.

Elements of $dx_1 \ldots x_k$ are constructed by element constructors c whose type must be of the form $(El\, e_1, \ldots, El\, e_n) \rightarrow El(dx_1 \ldots x_k)$ with e_i set pattern, i. e., only generated from those variables and arbitrary d's, but later the x_i can be any element of *Set*.

Rules for defined constants f have to respect the arity that is expressed by the prescribed type of f. They have the form $f\vec{p} = s$ with s beta-normal and p_i a constructor pattern: only built from variables, *fun* and c's. All substitution instances rewrite!

Rough Sketch of the System

The types are *Set* (the universe of data types), the type $El\,t$ of elements of a type code, represented by a term t of type *Set*, and the dependent function type (of the framework).
Elements of *Set* come from the Π (on the level of data types) and from the user-defined parameterized data types d. Elements of Π types are constructed by help of a primitive constant *fun*.
Elements of $dx_1 \ldots x_k$ are constructed by element constructors c whose type must be of the form $(El\,e_1, \ldots, El\,e_n) \longrightarrow El(dx_1 \ldots x_k)$ with e_i set pattern, i. e., only generated from those variables and arbitrary d's, but later the x_i can be any element of *Set*.
Rules for defined constants f have to respect the arity that is expressed by the prescribed type of f. They have the form $f\vec{p} = s$ with s beta-normal and p_i a constructor pattern: only built from variables, *fun* and c's. All substitution instances rewrite!

Rough Sketch of the System

The types are *Set* (the universe of data types), the type $El\ t$ of elements of a type code, represented by a term t of type *Set*, and the dependent function type (of the framework).

Elements of *Set* come from the Π (on the level of data types) and from the user-defined parameterized data types d. Elements of Π types are constructed by help of a primitive constant *fun*.

Elements of $dx_1 \ldots x_k$ are constructed by element constructors c whose type must be of the form $(El\ e_1, \ldots, El\ e_n) \rightarrow El(dx_1 \ldots x_k)$ with e_i set pattern, i. e., only generated from those variables and arbitrary d's, but later the x_i can be any element of *Set*.

Rules for defined constants f have to respect the arity that is expressed by the prescribed type of f. They have the form $f\vec{p} = s$ with s beta-normal and p_i a constructor pattern: only built from variables, *fun* and c's. All substitution instances rewrite!

Rough Sketch of the System

The types are *Set* (the universe of data types), the type $El\, t$ of elements of a type code, represented by a term t of type *Set*, and the dependent function type (of the framework).
Elements of *Set* come from the Π (on the level of data types) and from the user-defined parameterized data types d. Elements of Π types are constructed by help of a primitive constant *fun*.
Elements of $dx_1 \dots x_k$ are constructed by element constructors c whose type must be of the form $(El\, e_1, \dots, El\, e_n) \rightarrow El(dx_1 \dots x_k)$ with e_i set pattern, i. e., only generated from those variables and arbitrary d's, but later the x_i can be any element of *Set*.
Rules for defined constants f have to respect the arity that is expressed by the prescribed type of f. They have the form $f\,\vec{p} = s$ with s beta-normal and p_i a constructor pattern: only built from variables, *fun* and c's. All substitution instances rewrite!

Neighbourhoods

Atomic neighbourhood (p. 40): $\Delta \xrightarrow{[p/x]} \Gamma$ is written if $x : A$ is declared in Γ and Δ contains instead the type declarations of the variables that occur free in p so that p would receive the type A.

Compound neighbourhood: do this for several patterns.

The requirement on rules in order to have subject reduction is natural: Take the argument types of the type of f, form a context out of it, find the Δ of the compound neighbourhood w. r. t. $\vec{p}$. In Δ, the term s must have the target type of f, instantiated on $\vec{p}$.

Despite the natural idea, establishing subject reduction is quite some work and also needs a "strong generation lemma": If $\Gamma \vdash \lambda x.v : T$ then $T \equiv (x : U) \rightarrow V$ and $\Gamma, x : U \vdash v : V$.

Neighbourhoods

Atomic neighbourhood (p. 40): $\Delta \xrightarrow{[p/x]} \Gamma$ is written if $x : A$ is declared in Γ and Δ contains instead the type declarations of the variables that occur free in p so that p would receive the type A.

Compound neighbourhood: do this for several patterns.

The requirement on rules in order to have subject reduction is natural: Take the argument types of the type of f, form a context out of it, find the Δ of the compound neighbourhood w. r. t. $\vec{p}$. In Δ, the term s must have the target type of f, instantiated on $\vec{p}$.

Despite the natural idea, establishing subject reduction is quite some work and also needs a “strong generation lemma”: If $\Gamma \vdash \lambda x.v : T$ then $T \equiv (x : U) \to V$ and $\Gamma, x : U \vdash v : V$.

Neighbourhoods

Atomic neighbourhood (p. 40): $\Delta \xrightarrow{[p/x]} \Gamma$ is written if $x : A$ is declared in Γ and Δ contains instead the type declarations of the variables that occur free in p so that p would receive the type A.

Compound neighbourhood: do this for several patterns.

The requirement on rules in order to have subject reduction is natural: Take the argument types of the type of f, form a context out of it, find the Δ of the compound neighbourhood w. r. t. $\vec{p}$. In Δ, the term s must have the target type of f, instantiated on $\vec{p}$.

Despite the natural idea, establishing subject reduction is quite some work and also needs a "strong generation lemma": If $\Gamma \vdash \lambda x.v : T$ then $T \equiv (x : U) \to V$ and $\Gamma, x : U \vdash v : V$.

A remark on the reducibility predicate

There is an inductive definition when $RED_{Set}(t)$ holds. It has a finitely branching case when t reduces to a $d\vec{t}$ and an infinitely branching case when t reduces to a Π-type $\Pi t_1 t_2$. When $RED_{Set}(t)$ holds, an inductive definition is given when $RED_{El\,t}(u)$ holds. In the second case above, the branching is over the whole extension of $RED_{El\,t_1}$.

The definition of $RED_{El\,t}(u)$ is "introduction-based" and therefore enforces that if $fun\,v$ is reducible in its Π-type, then v is reducible in the respective function type of the framework (Lemma 3.4.7) and if $c\vec{u}$ is reducible in $d\vec{t}$, then the u_i are reducible in their respective argument types (Lemma 3.4.9). This is extended to the notions of neighbourhood in Lemma 3.6.4 and Lemma 3.6.5.

A remark on the reducibility predicate

There is an inductive definition when $RED_{Set}(t)$ holds. It has a finitely branching case when t reduces to a $d\vec{t}$ and an infinitely branching case when t reduces to a Π-type $\Pi t_1 t_2$. When $RED_{Set}(t)$ holds, an inductive definition is given when $RED_{El\,t}(u)$ holds. In the second case above, the branching is over the whole extension of $RED_{El\,t_1}$.

The definition of $RED_{El\,t}(u)$ is "introduction-based" and therefore enforces that if $fun\,v$ is reducible in its Π-type, then v is reducible in the respective function type of the framework (Lemma 3.4.7) and if $c\vec{u}$ is reducible in $d\vec{t}$, then the u_i are reducible in their respective argument types (Lemma 3.4.9). This is extended to the notions of neighbourhood in Lemma 3.6.4 and Lemma 3.6.5.

Call relation

$(f, \vec{t})$ calls $(g, \vec{v})$ means t_i reduces in finitely many steps to $p_i\gamma$ (with the same substitution γ), t_i has a normal form and there is a rule $f\,\vec{p} = s$ and $g\,\vec{u}$ is a subterm of s and $v_j = u_j\gamma$.

The crucial condition is that the "calls" relation is wellfounded. The Key lemma 3.6.6 on page 82 has the most complicated proof of the whole thesis: It requires in addition that the types of the defined constants f are reducible and shows then that the f's themselves are reducible. This is done by wellfounded induction on the call relation. It heavily profits from the characterization of typing for normal terms (the right-hand sides) that allows to prove inductively a suitable generalization (3.18 on page 83) and so treats variables that are set free by the subterm operation .

Call relation

$(f, \vec{t})$ calls $(g, \vec{v})$ means t_i reduces in finitely many steps to $p_i\gamma$ (with the same substitution γ), t_i has a normal form and there is a rule $f\vec{p} = s$ and $g\vec{u}$ is a subterm of s and $v_j = u_j\gamma$.

The crucial condition is that the "calls" relation is wellfounded. The Key lemma 3.6.6 on page 82 has the most complicated proof of the whole thesis: It requires in addition that the types of the defined constants f are reducible and shows then that the f's themselves are reducible. This is done by wellfounded induction on the call relation. It heavily profits from the characterization of typing for normal terms (the right-hand sides) that allows to prove inductively a suitable generalization (3.18 on page 83) and so treats variables that are set free by the subterm operation .

Outline

- The Key lemma is the deepest single element of the thesis.
- Theorem 4.2.1 (p. 91) that size-change termination implies wellfoundedness of the call relation is not very technical but rewarding since it shows how well prepared the situation is for this result.
- Theorem 5.2.2 assumes wellfoundedness of the call relation and gives a procedure to subsequently add two sets of defined constants and rules. The rules must not work on the respective other constants. The second set of constants must not be mentioned by the first sets of constants/rules. Then, if the procedure succeeds, one obtains a reducible system. The proof is broken down into more than 50 small steps, with references to results scattered throughout the thesis – a veritable invitation to read the thesis. The big challenge to extend this result is described on p. 103 (always under the assumption that the call relation is wellfounded!).

- The Key lemma is the deepest single element of the thesis.
- Theorem 4.2.1 (p. 91) that size-change termination implies wellfoundedness of the call relation is not very technical but rewarding since it shows how well prepared the situation is for this result.
- Theorem 5.2.2 assumes wellfoundedness of the call relation and gives a procedure to subsequently add two sets of defined constants and rules. The rules must not work on the respective other constants. The second set of constants must not be mentioned by the first sets of constants/rules. Then, if the procedure succeeds, one obtains a reducible system. The proof is broken down into more than 50 small steps, with references to results scattered throughout the thesis – a veritable invitation to read the thesis. The big challenge to extend this result is described on p. 103 (always under the assumption that the call relation is wellfounded!).

- The Key lemma is the deepest single element of the thesis.
- Theorem 4.2.1 (p. 91) that size-change termination implies wellfoundedness of the call relation is not very technical but rewarding since it shows how well prepared the situation is for this result.
- Theorem 5.2.2 assumes wellfoundedness of the call relation and gives a procedure to subsequently add two sets of defined constants and rules. The rules must not work on the respective other constants. The second set of constants must not be mentioned by the first sets of constants/rules. Then, if the procedure succeeds, one obtains a reducible system.
  The proof is broken down into more than 50 small steps, with references to results scattered throughout the thesis – a veritable invitation to read the thesis. The big challenge to extend this result is described on p. 103 (always under the assumption that the call relation is wellfounded!).

- The Key lemma is the deepest single element of the thesis.
- Theorem 4.2.1 (p. 91) that size-change termination implies wellfoundedness of the call relation is not very technical but rewarding since it shows how well prepared the situation is for this result.
- Theorem 5.2.2 assumes wellfoundedness of the call relation and gives a procedure to subsequently add two sets of defined constants and rules. The rules must not work on the respective other constants. The second set of constants must not be mentioned by the first sets of constants/rules. Then, if the procedure succeeds, one obtains a reducible system. The proof is broken down into more than 50 small steps, with references to results scattered throughout the thesis – a veritable invitation to read the thesis. The big challenge to extend this result is described on p. 103 (always under the assumption that the call relation is wellfounded!).