Coherence Checking of Coercions in Plastic*

Paul Callaghan
Department of Computer Science, University of Durham,
South Road, Durham DH1 3LE, U.K.
`http://www.dur.ac.uk/CARG`
`P.C.Callaghan@durham.ac.uk`

Abstract

Coercive Subtyping is a conceptually simple but powerful framework for subtyping in type theory. Recent work in Durham by Luo et al. on Coercive Subtyping has been partially implemented in the proof assistant 'Plastic'. Plastic is an implementation of a variant of Martin-Löf's LF with inductive types, universes, and coercive subtyping. After discussing background theory and then the implementation of coercions, this paper considers the important issue of coherence. Where several coercions are applicable, coherence will guarantee that the computational result is the same in all cases.

Checking coherence is a difficult problem for implementations. Previous implementations have tackled this problem by imposing restrictions on the coercions that may be used, which also limits the power of coercions, eg making them harder to use with definitions. This paper considers ways around the problem in a less restricted setting. One possibility is to only check coherence of the coercions *actually used*, rather than trying to check it globally over all possible coercions whether they are used or not; this technique we call "dynamic coherence checking".

1 Introduction

In recent papers, Luo defines Coercive Subtyping as an abbreviational mechanism in a logical framework [Luo97, JLS98, Luo99]. Other work has introduced a coercion mechanism in to a particular type theory, for example: Bailey in Lego's ECC [Bai98] and Saïbi in Coq's CIC [Saï99]. But, it is argued that the most natural interpretation of Coercive Subtyping is as a general mechanism independent of particular type theories, hence should be introduced via a *logical framework*. The framework theory we use is a variant of Martin-Löf's logical framework[1], the main difference being that lambda bindings are explicitly typed [Luo94]. For brevity, the variant of LF with explicitly typed lambda abstractions ($[x:K]t$ vs $[x]t$) will henceforth be called LF, and the Martin-Löf form will be named explicitly. Coercive subtyping has many important applications, including:

*This work is supported by the project "Subtyping, Inheritance, and Reuse", funded by UK EPSRC (GR/K79130, see `http://www.dur.ac.uk/CARG/sir.html`).

[1] See [NPS90] for details of the Martin-Löf's framework.

- Formalisation of mathematics in type theory [Bai98, Saï99], where coercions make manipulation of structured mathematical objects more convenient and natural.
- General programming: many uses, e.g. coercions allow overloading of names, such as seen in programming languages where class methods may be used on values of different types. We are currently investigating the incorporation of coercive subtyping in to the Haskell type system [LCK00], and applications in strong functional programming [KLM99].
- Universes in type theory: universes in the Tarski style are introduced as part of UTT in [Luo94]; such universes are semantically clearer than Russell-style as used in ECC and implemented in Lego, but less convenient to use. Declaring the lifting functions between universes as coercions makes Tarski-style universes more convenient [CL00b].
- Natural Language: several uses for coercions have been found in our recent work on mathematical language [LC98b]. Coercions also provide an elegant analysis of certain phenomena of lexical semantics which have been problematic for other approaches [LC98a].

The metatheory of this approach to subtyping has been studied in other papers [Luo97] [JLS98] [SL98a] [SL98b], and the approach was recently extended to cover "dependent coercions" [LS99a] (discussed in section 2.4.4). A significant problem for this approach is the issue of coherence checking. The coherence property guarantees that where several coercions are applicable at in some derivation, then it does not matter which is chosen because the computational result will be the same.

Checking coherence is difficult both in the theory and in the implementation. Firstly, use of an intensional equality means that we have no simple way of checking the equivalence of two coercions within the system. Secondly, with the richer forms of coercion in Luo's system, checking is undecidable due to the number of possibilities implied. Some form of meta-level reasoning may be required. We note that coherence checking may not be as severe a problem in other approaches, such as record-based subtyping. But Luo's system is clearly more general than these; the question is how to make use of the generality without paying the price of being unable to guarantee coherence.

1.1 Related Work

Coercive subtyping is one approach to subtyping, and may be traced to the work of Breazu-Tannen et al. [BCGS91]. General approaches include those inspired by object oriented programming (eg [Car88]) and much work related to Girard's system F (e.g. [LMS95, CG97]). Of the work based within type theory, there is also work on record-based subtyping (e.g. [Tas97, Bet97]), or constructor-based subtyping [BF99].

In the area of proof development based on type theory, Luo's approach to coercive subtyping as a framework notion is general and uniform. Luo also comments that coercive subtyping is an attractive approach when making heavy use of inductive types; some earlier approaches cannot be adapted to these circumstances. A coercions-based explanation of universes in the Tarski style (as used in UTT) is a good example of the benefits of coercive subtyping in the presentation of a type theory (introduced in [Luo99], and discussed in [CL00b]); compared to Luo's earlier version in ECC that used Russell-style universes and a built-in cumulativity relation [Luo94], the new presentation is much clearer. See section 5 of [Luo99] for a more complete discussion of approaches to subtyping.

Forms of coercive subtyping have been implemented in Lego's ECC by Bailey [Bai98] and in Coq's CIC[2] by Saïbi [Saï99], with various restrictions to make the theory implementable and efficient. Note that coherence checking is decidable (and has been implemented) for these restricted systems. Whilst the implementations are a significant improvement in convenience for use of type theory (consider the large proof developments in Bailey's and Saïbi's theses), and provide a fair amount of flexibility, the restrictions do sometimes lead to frustrations in what their coercions cannot do, for example with definitions, or with more powerful coercions.

1.2 Aims and Organisation

The purpose of this paper is to document the current state of coercions in Plastic, and to discuss a significant problem for all implementations of coercive subtyping. 'Plastic' is a tactic-based proof assistant for LF, with metavariables, inductive types, and universes. The technical interest in Plastic arises because it is the first implementation of LF-based coercive subtyping in the style of [Luo99], in contrast to previous implementations which are based on, and tied to, specific type theories. A further justification for Plastic is to evaluate the approach of using a framework system as a platform for implementing domain-specific reasoning tools; coercive subtyping is a key part of the approach, since it is believed that it will simplify the communication between an object type theory (and a tool based on it), and the underlying framework type theory. Thus we can evaluate the impact of coercions in diverse applications of type theory.

The relevance for theory concerns the *implementability* of Luo's form of coercive subtyping. The prototype (Plastic) does not impose the restrictions that are made in Lego and Coq, and implements forms of coercion which are not allowed in those systems. How successful this can be is an open question; we need to study techniques for efficiently implementing the wider system, and to understand how the benefits balance against the costs.

Organisation The next section introduces background theory for Coercive Subtyping, and considers the property of Coherence and several deeper issues. The current implementation of coercions in Plastic is surveyed next, covering the basic techniques. The specific problem of implementing coherence checking is considered in the final section.

[2]Saïbi's techniques are presented as an extension of PTSs, of which CIC is a close relative.

2 Coercive Subtyping

We first present the framework theory LF, outline the main ideas of Coercive Subtyping, then discuss several deeper, technical aspects of coercions.

2.1 The LF framework

LF provides the basic components of dependent type theories, such as a simple notion of dependent function space, but little more. One may introduce more complex concepts as needed for a specific task by declaring appropriate constants etc. Please see any of [Luo94] [Luo99] [CL00b] for the rules of the core LF system. Important points to note include:

- 'Types' in the framework are called 'kinds' in order to avoid confusion with types in an object (i.e., user-specified) type theory.
- There is a special kind **Type** which represents the conceptual universe of types of the object type theory. Given a type $A : \mathbf{Type}$, there is a kind $El(A)$ corresponding to the collection of objects of type A. Thus El lifts object theory types into the kind system. We refer to such kinds as *El-kinds*.
- In addition to usual judgement form $\Gamma \vdash x : T$, LF has a form to represent definitional equality, e.g. $\Gamma \vdash k = k' : K$, which says k and k' are equal objects of kind K. Definitional equality is a framework notion, and is used to define object type theories. Definitional equality is also a suitable mechanism for representing abbreviations, such as definitions and coercive subtyping.
- Functions in LF are called *functional operations*; a subset of these are λ-abstractions, written $[x : K]k$ (contrast this to Martin-Löf's untyped $[x]k$). The kinds of functional operations are called *dependent products*, written $(x : K)K'$, or as $K \to K'$ if x does not occur free in K'. Reduction of functional operations applied to objects in their domain corresponds to β-reduction.
- Dependent products are not equivalent to Π-types in languages like ECC. A significant difference is the η rule for functional operations. The η rule makes little sense as a computation rule, so isn't necessary in an object theory [Luo99] (p. 108). So, equality in object theories must be computational.
- Object theories are specified by declaring new constants and then extending the definitional equality to specify the behaviour of the constants.

Core LF is supplemented by a scheme for inductive types (induction via strictly positive operators), which supports addition of the usual types for programming. One may also add universes, by declaring appropriate constants and extending the definitional equality. The type theory UTT[3] is specified in this way [Luo94, Gog95].

[3] UTT is a clearer version of the theory ECC which is implemented in Lego [LP92]; differences include use of Tarski style universes (as opposed to Russell style cumulativity).

2.2 Coercive Subtyping in LF

As noted above, definitional equality supports the introduction of coercive subtyping as an abbreviation mechanism in the framework. Particular object theories can then inherit subtyping by virtue of their specification in the framework. Apart from the convenience, this approach also allows us to study various forms of subtyping in a uniform framework, e.g. Bailey's notions of Π-coercion and kind coercion [Bai96] are expressible as instances of a single underlying concept.

A coercion is a functional operation $c : K_0 \to K$, which lifts an object of kind K_0 to kind K; the existence of a particular coercion c is represented by the notation $K_0 <_c K$, and an unnamed coercion by $K_0 < K$. The introduction of coercions in to the system is discussed in section 2.5. The meaning of coercion use is expressed via the *coercive definition rule*:

$$\frac{f : (x{:}K)K' \quad k_0 : K_0 \quad K_0 <_c K}{f(k_0) = f(c(k_0)) : [c(k_0)/x]K'}$$

This says that term $f(k_0)$ abbreviates and is definitionally equal to a term where the coercion is made explicit, namely $f(c(k_0))$, when a coercion c exists to lift object k_0 to the kind expected by the functional operation f. Notice that coercions are only used in a context where the expected kind is known, i.e. where we know *both* K and K_0. The explicit typing on lambda abstractions supports this situation, by making all relevant type information easily available. Contrast this against the original Martin-Löf LF, with untyped λ-abstractions: type inference would need to find the type of each bound variable before determining which coercion is required. (A similar situation occurs for λ-abstractions when coercive subtyping is introduced in the Hindley-Milner system; see [LCK00], which considers the addition of coercive subtyping to Haskell [Has].)

A concrete example: if $f : (x : Vec_A(n))\ T$ expects a vector but $k_0 : List_A$ is a list, and there was a coercion $List_A <_{c_1} Vec_A(n)$ which converts lists to vectors, then term $f(k_0)$ would be well typed with kind T, and definitionally equal to the explicitly coerced form $f(c_1(k_0))$.

The full system contains other rules, e.g. expressing the well-typedness of coercion use, and there are several possible extensions to this basic scheme, as discussed in section 2.4. We first consider the important issue of coercion coherence.

2.3 The Problem of Coherence

In some part of a term, several coercions among the set declared in the system may be applicable. Intuitively, the property of coherence provides a guarantee that it does not matter which coercion is chosen: the computational result will be the same regardless. But how to state this intuitive condition precisely in (intensional) type theory is still an open question. The most basic and necessary condition is as follows ([Luo99] lists two additional minor conditions):

Definition 1 (coherence condition [Luo99]) *We say that the basic subtyping rules are* coherent *if [a system] has the following* coherence properties*:*

1. *If* $\Gamma \vdash A <_c B : \mathbf{Type}$ *and* $\Gamma \vdash A <_{c'} B : \mathbf{Type}$, *then* $\Gamma \vdash c = c' : A \to B$.

An alternative, more powerful version for dependent coercions is given in [LS99a], and is considered in section 2.4.4. The formulation above is intensional, in accordance with the underlying type theory. This can be problematic when checking coherence; for example, a coercion composed with its inverse may not be intensionally equal to the identity function for the relevant type. In an extensional theory [ML84], one can use an extensional notion of coherence, which removes the above problem with inverses. Extensional coherence is a subject which needs further study ([LS99a], section 5.1).

To summarise, how to state the coherence property is an open question. The process for checking coherence is difficult, and becomes undecidable with parametrized coercions or dependent coercions, because the set of coercions becomes potentially infinite. Obviously, it is a serious problem for practical implementations. We discuss attempts at solution in section 4.

2.4 Extensions of Coercive Subtyping

2.4.1 Subkinding for dependent product kinds

More flexibility can be gained by allowing subtyping to work across dependent products. The rules below allow (respectively) for subkinding on the range, subkinding on the domain, and subkinding on both the domain and range [Luo99]. Notice that the subkinding on domain is contravariant.

$$\frac{\Gamma, x{:}K_1 \vdash K_2\ \mathbf{kind} \quad \Gamma, x{:}K_1 \vdash K_2 <_c K_2'}{\Gamma \vdash (x{:}K_1)K_2 <_{[f:(x:K_1)K_2][x:K_1]c(f(x))} (x{:}K_1)K_2'}$$

$$\frac{\Gamma, x{:}K_1 \vdash K_2\ \mathbf{kind} \quad \Gamma, x{:}K_1' \vdash [c(x)/x]K_2 = K_2' \quad \Gamma \vdash K_1' <_c K_1}{\Gamma \vdash (x{:}K_1)K_2 <_{[f:(x:K_1)K_2][x:K_1']f(c(x))} (x{:}K_1')K_2'}$$

$$\frac{\Gamma, x{:}K_1 \vdash K_2\ \mathbf{kind} \quad \Gamma \vdash K_1' <_{c_1} K_1 \quad \Gamma, x{:}K_1' \vdash [c_1(x)/x]K_2 <_{c_2} K_2'}{\Gamma \vdash (x{:}K_1)K_2 <_{[f:(x:K_1)K_2][x:K_1']c_2(f(c_1(x)))} (x{:}K_1')K_2'}$$

This increases the applicability of coercions. For example, given a coercion from even numbers to natural numbers $\Gamma \vdash Even <_c Nat$, we have:

$$\Gamma \vdash T \to Even < T \to Nat$$

$$\Gamma \vdash Nat \to T < Even \to T$$

$$\Gamma \vdash Nat \to Even < Even \to Nat$$

$$\Gamma \vdash Nat \to Nat \to Even < Even \to Even \to Nat$$

Notice that the final example is a combination of the previous two. This form of subtyping is very useful for working with names in universes of parametrized inductive types [CL00b].

2.4.2 Parametrized Coercions

Intuitively, we may coerce between a vector of length n and of element type A to a list of element type A, e.g. a vector of natural numbers to the equivalent list. The empty vector is mapped to the empty list, and vector *cons* is mapped to list *cons*. This process is purely structural, and is not sensitive to the element type or to the length of the vector, so we can think of a family of such coercions, parametrized by A and n, as an abbreviation for all concrete instances.

$$\Gamma, A : \mathbf{Type}, n : Nat \vdash Vec(n, A) < List(A)$$

This form of parametrization does not cause many problems, and is implemented in Lego and Coq. But in general, parametrized coercions add complication for coherence checking, because of the unbounded possibilities represented. A good example is first projection from a Σ-type:

$$\Gamma, A : \mathbf{Type}, B : A \to \mathbf{Type} \vdash \Sigma(A, B) <_{\pi_1} A$$

The result type clearly depends on a parameter. We will consider the second projection in section 2.4.4.

2.4.3 Lifting Coercions over Parametrized Inductive Types

Given the coercion $Even <_c Nat$, and a polymorphic type of Lists, it is convenient to coerce a list of even numbers to the corresponding list of natural numbers. This step can be repeated, so we could next coerce a list of list of even numbers to a list of list of naturals, and so on. For lists, given $c_L(nil(A)) = nil(B)$ and $c_L(cons(A, a, l)) = cons(B, c(a), c_L(l))$, this means:

$$\frac{\Gamma \vdash A <_c B : \mathbf{Type}}{\Gamma \vdash List(A) <_{c_L} List(B) : \mathbf{Type}}$$

The process can be applied to other inductive types, and the lifting functions or "coercion rules" derived automatically for the class of inductive types allowed in LF ([Luo99], p. 123). Saïbi also mentions this extension of coercions in his thesis [Saï99], but has not implemented it in Coq. The derivation follows straightforwardly from the structure of the inductive types; hence the following result for Σ-types (ie. dependent pairs, $\Sigma : (A{:}\mathbf{Type})(B{:}(A)\mathbf{Type})\mathbf{Type}$):

$$\frac{\Gamma \vdash A < A' \quad \Gamma, x : A \vdash B(x) < B'(x)}{\Gamma \vdash \Sigma(A, B) <_{c_\Sigma} \Sigma(A', B')}$$

Notice that we have several choices of how to coerce when the inductive types have more than one parameter; a good example is Σ-types, where we can subtype by either the first element or by the second element. With type constructor Σ and constructor $pair_\Sigma$, we have two basic rules:

$$\frac{\Gamma \vdash B : (A)\mathbf{Type} \quad \Gamma \vdash B' : (A)\mathbf{Type} \quad \Gamma, x{:}A \vdash B(x) <_{c[x]} B'(x) : \mathbf{Type}}{\Gamma \vdash \Sigma(A, B) <_{c_1} \Sigma(A, B') : \mathbf{Type}}$$

$$\frac{\Gamma \vdash A <_c A' : \mathbf{Type} \quad \Gamma \vdash B' : (A')\mathbf{Type}}{\Gamma \vdash \Sigma(A, B') <_{c_2} \Sigma(A', B') : \mathbf{Type}}$$

where $pair_\Sigma{:}(A{:}\mathbf{Type})(B{:}(A)\mathbf{Type})(a{:}A)(b{:}B(a))\Sigma(A, B)$, $c_1(pair_\Sigma(A, B, a, b)) = pair_\Sigma(A, B', a, c[a](b))$, and $c_2(pair_\Sigma(A, B', a, b)) = pair_\Sigma(A', B', c(a), b))$. Note that we can use the fact that $\Sigma(A, B') = \Sigma(A, B' \circ c)$: **Type** (by the coercive definition rule) to abbreviate $\Sigma(A, B' \circ c)$ by $\Sigma(A, B')$.

In isolation, without any user-specified coercions, these derived coercions can be shown to be coherent ([Luo99], p. 124). Luo recommends adopting such rules as system defaults, introduced when the relevant inductive types are introduced, but acknowledges that users might prefer to use other coercions; he suggests that some sort of priority system could be used.

2.4.4 Dependent Coercions

A natural generalisation is to allow coercions from types to *families* of types [LS99a]. This increases the expressive power, and allows many more useful coercions, but at cost of additional complexity in the system, especially with coherence checking, which becomes undecidable due the extra checking entailed.

A simple example is coercion of lists to fixed-length vectors, where the size of the vector is $len(l)$, the length of the list l. The notation follows [LS99a], indicating a dependent coercion c whose range depends on its domain object l.

$$l : List_A \xrightarrow{c} Vec_A(len(l))$$

A second (parametrized) example is second projection for Σ-types, where the type of the result depends upon the first projection:

$$\Gamma, A : \mathbf{Type}, B : A \rightarrow \mathbf{Type} \vdash p : \Sigma(A, B) \xrightarrow{\pi_2} B(\pi_1(p))$$

Luo and Soloviev show that all dependent coercions on types can be written as a non-dependent coercion composed with π_2 [LS99b].

The subkinding rules for dependent products can be extended for dependent coercions, thus generating dependent coercions over functional operations. How far they should be generalised is an interesting question; for example, are there serious practical applications which justify the extra complexity for the metatheory? We can use Plastic to help investigate such questions.

A modified coherence condition for dependent coercions is given in [LS99b], and shown below. The significant change is that A must be non-empty. When x does not occur free in B or B', we get the coherence condition of section 2.3.

Definition 2 (coherence condition [LS99b]) *A set R of basic subtyping rules is* coherent *if the following is true in [the system]:*

- *If* $\Gamma \vdash x{:}A \xrightarrow{c} B[x] : \mathbf{Type}$, $\Gamma \vdash x{:}A \xrightarrow{c'} B'[x] : \mathbf{Type}$, $\Gamma \vdash a{:}A$, *and* $\Gamma \vdash B[a] = B'[a] : \mathbf{Type}$
- *Then* $\Gamma \vdash c(a) = c'(a) : B[a]$

The extra power raises new problems. If the list-to-vector coercion c is (post-) composed with its complement vector-to-list (say, c'), then the result is as follows:

$$c \circ c' : (n : Nat)(v : Vec_A(n))Vec_A(len(c'(v))$$

Intensionally, this is not equivalent to the identity function, so this composition could be taken as a coercion between two different types. As mentioned before, an extensional notion of coherence may be required here. Such coercions, which are not intensional identities, could have applications in proofs on algebraic equations, for example by representing potential rewrites. A nice example is with commutative operators, thus removing the artificial distinction between $T(a \oplus b)$ and $T(b \oplus a)$. Further research is needed to understand the implications.

2.5 Introducing Coercions in to the System

Luo et al have not discussed this practical aspect in detail; essentially, it is assumed that a set $\mathcal{R}$ of "basic subtyping rules" has been declared by the user, over which the coherence property holds. Each rule represents a judgement $\Gamma \vdash A <_c B$. Further coercions come from the lifting of coercions over inductive types (section 2.4.3); Luo suggests that such lifted coercions could be taken as "system defaults" on account of their wide applicability. But it is it is possible for these to conflict with user-specified rules. In this situation Luo suggests that user-specified coercions should take priority over the default rules. There has been no formal definition of this priority.

Lego and Coq approach this problem by building up the coercion set incrementally, as each new coercion is declared. If a proposed coercion fails the coherence checking, it is rejected. Otherwise it is accepted, and stored in the context, perhaps with new coercions which can be generated by composition with earlier coercions. The implementations are following the normal use of context: a coercion declaration is thus similar to a hypothesis or definition. Coercions may also depend on previous hypotheses or definitions in the context; this is in contrast to them being a separate or 'global' concept.

The natural extension of this is to consider that coercion use is fully context-dependent, so different sets of coercions may apply in different contexts. We might also consider a notion of binding a coercion in a subterm, in analogy to how one λ-binds variables.

A related issue concerns the status of some forms of coercion. One might argue that coercions like π_1 are properly a family of coercions, and should be regarded as a convenient way (for the user) to generalise over similar instances, rather than a coercion *per se*. Another way of saying this is to regard a parametrized coercion as a template for concrete coercions. A 'real' or concrete coercion is generated from the template when values are found for all its parameters, so that the concrete coercion can be inserted in a term. This discussion is very relevant to the representation and manipulation of coercions in an implementation. For example, the dynamic coherence checking of section 4.1 takes this view very strongly: that only the concrete coercions that are actually used are candidates for coherence checking.

2.6 Understanding of Coercions

Coercive Subtyping offers a great deal of flexibility for use of type theory, but it also raises difficult questions for understanding. We can have many coercions active, and also are not prevented from declaring coercions which have non-intuitive actions. In this case, understanding of a term in type theory is dependent upon being aware of all coercions implicitly used in the term, and understanding their precise effects - both in isolation and *combined* with other possible coercions. Clearly, this leads first to obscurity and then to errors in one's use of type theory.

One way to reduce this problem is to impose semantic conditions on coercions, so that only intuitively understandable coercions are used. Record-based subtyping is a good example, where coercions just forget structure in records to give smaller records; such coercions are easier to understand. But in the context of coercive subtyping, how to mechanise such constraints is not obvious. We do not offer solutions to the problems; the intention here is to highlight this aspect of coercions. It is part of the pragmatic aspects of coercive subtyping which need to be further studied.

3 Plastic and its implementation of Coercions

Plastic is an implementation of LF with inductive types, universes, and coercive subtyping. It is a proof assistant in the style of Lego [P^+b] and Coq [Coq], thus uses tactics rather than direct manipulation as in Alfa [CC99]. It can run on most modern platforms, and is best used with Aspinall's Proof General interface for `xemacs` [Asp]. More information on Plastic can be found in [CL00a] [CL00b] [CL98], or see the URI `http://www.dur.ac.uk/CARG/plastic.html`.

The core of Plastic is an abstract syntax of LF, with algorithms for manipulating it, plus a standard algorithms for type inference and convertibility. It follows [Hue89] in many details; one difference is that the Constructive Engine does not implement η conversion, which we require in LF (we use the technique due to Coquand [Coq91]). Another difference is the implementation language: we use the non-strict functional language Haskell [Has]. One might think that use of lazy languages implies slower programs, but with current technology (e.g., the GHC compiler [P^+a]), we have no reason to regret our choice. The laziness also allows us to write much of the system in a straight-forward way, eg without needing explicitly to delay substitutions.

Plastic also implements inductive families [CL00a] and universes [CL00b], the theory of which is presented in chapter nine of the monograph [Luo94]. There is a simple metavariable facility, based on first-order unification, which provides a notion of refinement proof. Deliberately, there is no conventional goal-directed proof mode; instead there are just metavariables and tactics for manipulating them. The remainder of this section will present the current implementation of coercive subtyping. We do not give detailed examples of coercion use here; Plastic supports most of the examples[4] in previous papers, e.g. [Luo99]. Coherence checking will be considered in more detail in the next section.

[4] We encourage people to experiment with Plastic themselves. We plan to release executables with a library in the summer of 2000, time permitting, for Solaris and Linux platforms.

3.1 Implementation of Coercions

Coercions are inserted where required during type inference, when a functional operation f with domain K is applied to an object v of different kind K' and a coercion $c : K' \to K$ can be found. The result is $f(c(v))$; the coercion application is by default not shown, but behaves as a conventional application node under convertibility tests. Plastic provides options to show the location of coercions by a single mark ! or to show the coercion in full but using a textual effect to distinguish it from conventional subterms.

Note that this insertion style, whilst similar to the techniques in [Bai98, Saï99], is different from the theoretical presentation where the intended notion of computation is *typed reduction* (e.g., [Luo99], section 3.2.4). This is acceptable, since work on the metatheory, specifically the conservativity proofs [SL98b], guarantees that every derivation containing coercions can be transformed to a derivation in the coercion-free calculus.

The most important stage of coercion implementation is *computation*, of determining the coercion term to be inserted – if one exists. The process for declaring coercions and the representations used are also important. The relative importance of the stages depends on the form of coercions being used; for example, the simplifying restrictions in Bailey's implementation in Lego allow for a simpler computation step than is used in Plastic. In the following sections, we discuss the stages relevant to Plastic in some detail.

3.2 Matching of Types in Coercions

A significant restriction in existing implementations [Bai98, Saï99] is that in their algorithm, types are matched syntactically *by name*, rather than by convertibility. This restriction is mainly motivated by concerns of efficiency, since full convertibility tests are expensive.

Typically, syntactic matching by name allows indexing of coercions by the head type constructors in their domain and range types. Such type constructors act as convenient labels in a *graph* of coercions, convenient because comparison between labels (e.g. in look-up) is a cheap test. The graphs thus support efficient lookup of a coercion between two types, and checking of coherence by identification of distinct paths between two nodes in the graphs. The theory underlying the Coq implementation [Saï99] emphasises the importance of the names of type constructors, and the concept of a coercion graph is used in both previous implementations.

As noted above, the restriction to syntactic matching is too inflexible for widespread use of coercions. For example, one could have a coercion defined on a type $T[x, y]$, and name a sub-case where x and y are identical, ie. $U[x] = T[x, x]$. In the Coq and Lego implementations, names $U[x]$ and $T[x, x]$ will be considered different, and hence one would need to explicitly extend the coercion to the special case.

The notion of a finite graph is also problematic because we allow a potentially infinite number of coercions, for example the arbitrary nesting of coercions between list types (by coercion rules). We also allow patterns of the form:

$$A : \mathbf{Type}, \quad C : \mathbf{Type} \to \mathbf{Type}, \quad D : \mathbf{Type} \to \mathbf{Type} \vdash C(A) <_{c_A} C(D(A))$$

where C and D are type constructors. This coercion generates a set $C(D^i(A))$ where i is non-negative. A concrete instance of this is a coercion from an n-ary tree to an $n+1$-ary tree (by adding an empty subtree to each node), which could be used to convert a 1-ary tree (isomorphic to a list) to a 3-ary tree etc[5].

A similar pattern occurs with Σ-types, where we may want to use first projection π_1 on a nested Σ value, eg

$$\Sigma(\ldots\Sigma(A, B_1)\ldots, B_n) <_{\pi_1{}^n} A$$

where $\pi_1{}^n$ is the first projection applied n times, each time suitably parametrized. Clearly this coercion will be useful for working with record values. But it is not possible to know in advance how deep a nesting of Σ-types will be used, and hence not possible to calculate the coercions in advance[6].

For such reasons, and to allow greater experimentation, Plastic does not implement this restriction of matching by name only. Instead, matching of coercions uses the internal convertibility test. Remember that such experimentation is a central design goal of Plastic, and we are not too concerned about efficiency at present. Using convertibility also makes it harder to build a convenient graph data structure with efficient indexing. At present, Plastic does not attempt to build such structures, relying on a simpler technique of trying a match against each coercion in the list of currently active coercions[7].

3.2.1 Coercion Declaration

Coercions are introduced by declaring a term to be a coercion. An internal name for this coercion is generated automatically, for consistency. One may give a type for this term, e.g. for documentation, but it is optional. An example is given below:

```
> [map : (A,B:Type) (A -> B) -> List A -> List B ]
> Coercion Parameters [A,B:Type]
>          Prerequisites [f:A->B]
>           = map A B f : List A -> List B;
```

Parameters of a coercion are preceded by an explicit `Parameters` keyword. "Coercion rules", where higher coercions can be generated from simpler coercions (see section 2.4.3), are further distinguished by the keyword `Prerequisites` marking

[5]This example has been investigated by Alex Jones (at Durham).

[6]In contrast, Bailey [Bai98] has to define each projection on his algebraic structures separately (eg group to monoid, monoid to semigroup etc). His approach has the advantage, however, of indicating semantically what each step means.

[7]It may be possible to encode types in some suitable index type which does permit efficient retrieval of coercions, e.g. using hashing techniques on terms to provide a rough match, or using 'negative' information to prevent matches that will fail, such as tests on parametrized coercions that can be done before a correct instantiation is found. We believe that efficient matching of types is a subject that deserves more work, and will contribute to type theory technology.

the parameters which *must* be coercions themselves. The example above declares mapping on list as a coercion (in our terminology, properly a coercion rule), with parameters for the element types, and a prerequisite that $f : A \to B$ is itself a coercion.

The prerequisite coercions are temporarily declared as coercions whilst Plastic is processing a coercion declaration, to allow the prerequisites to act in the definition. This facility is used in the declaration of subtyping over the second element in a Σ-type (section 2.4.3). Instead of having to write some of the `B'` as `[x:A]B (f x)`, we can write the following directly (`E_Sigma` is the elimination operator for Σ-types):

```
> Coercion
>   Parameters [A':Type][B':A'->Type][A:Type]
>   Prerequisites [f:A -> A']
>    = E_Sigma A B' ([_:Sigma A B']Sigma A' B')
>                   ([x:A][y:B' x]sigma A' B' (f x) y);
>    : Sigma A B' -> Sigma A' B' ;
```

3.2.2 Internal Representations

Coq and Lego both construct graphs of coercions, based on the names of types. As explained above, Plastic does not do this. The current implementation just stores coercions as a list of declarations. This list is (expensively) manipulated each time a coercion is required.

One refinement has been implemented. To control the search for coercions, we calculate compositions of simple coercions (i.e., those without coercion parameters), in an attempt to generate reasonable closure of this set. On declaration of a coercion c, in the presence of existing coercions $\mathcal{C}$, the current algorithm forms compositions[8]

$$c_i \circ c,\ c \circ c_i, \qquad (c_i \in \mathcal{C})$$

$$c_i \circ c \circ c_j, \qquad (c_i,\ c_j \in \mathcal{C})$$

and adds all the compositions which are well-typed as new coercions. The new coercions are not then subject to the same process, because in some cases (see the example of $C(D^i(A))$ above), this process would not terminate.

Clearly, this is just a heuristic measure which provides convenient compositions for most cases. Potential compositions which are missed by this heuristic can be added explicitly by hand, if required. The issue of non-termination is what motivated this step; whilst searching for a coercion, searching possible compositions of coercions will not terminate. This is reflected by the failure (to date) to eliminate transitivity in meta-theoretic studies of LF with Coercions.

The process can generate many useless coercions, and these are extra coercions that need to be tested when next searching for a particular coercion. Such coercions are, however, a consequence of transitivity in Coercive Subtyping. Some limited coherence checking is done on the declared coercion and its resulting valid

[8] The algorithm makes use of the metavariable mechanism to calculate instantiations of parametrized coercions, as explained in section 3.2.3.

compositions, including filtering out the coercions which are intensionally equal to identity functions; this will be explained in section 4.

3.2.3 Coercion Computation

When type synthesis detects a functional operation of kind $(x : K_1)K$ applied to a value v of an incompatible kind K_2 it needs to find a coercion c of kind $(x : K_2)K_1[x]$ such that $c(v) : K_1$. This is the key step in the current implementation.

As mentioned above, there is no convenient graph data structure; the current algorithm sequentially (and expensively) tests each entry in the list of declared coercions for a suitable match, using the internal convertibility test. In the case of parametrized coercions, this includes attempting to find a suitable instantiation. For coercion rules, we must also search for a prerequisite coercion (whose type we know), using the same process again. Notice that this latter process will terminate only for the standard rules discussed in section 2.4.3, because at each step the types we need to coerce between are structurally smaller.

Instantiations of parametrized coercions are calculated with the metavariable mechanism, which is based on first-order unification. This is similar to using the `Refine` tactic in proofs, except that the coercion search *may not* terminate with unsolved metavariables. That is, inserted coercions must be fully concrete terms, and not contain unsolved metavariables. The techniques used appear similar to that described in [Saï99]; Plastic's version was implemented independently, before we encountered this work. Saïbi also studies properties of his algorithms.

As mentioned above, the search restricts itself to the list of declared coercions. It does not attempt to generate new coercions on the fly, e.g. by composition (i.e., using the transitivity rule), or by arbitrary application of the coercion rules (e.g. *map* over lists), since such processes will not terminate.

4 Implementation of Coherence Checking

Coherence checking is a significant theoretical issue for Coercive Subtyping, and is critical for practical implementations. The theory outlined in [Luo99] and subsequent papers allows potentially infinite sets of coercions, making coherence undecidable through the unbounded number of possibilities to check. Previous implementations (ie. Bailey and Saïbi in Lego and Coq) have imposed restrictions on the theory being implemented to remain tractable. Several ideas for tackling the problem in an unrestricted system were mentioned in [Luo99] (p. 128) and [LS99a], but not explored in detail. This section focusses on "dynamic checking". Other current ideas are:

- "Meta-arguments": the user guarantees (meta-theoretically, i.e. outside the system) that a set of coercions is internally coherent. Whilst only these coercions are used, there is no need to check coherence. Only when different sets of coercions interact does any checking need to be done. Automatically generated coercions, as discussed in section 2.4.3, may provide another source of internally coherent coercion. This partitioning of coercions may provide other benefits, in the sense of the divide-and-conquer heuristic.

- "Asymmetric coercions": allowing coercions from $List(A)$ to $List(B)$, for arbitrary A and B, may be too powerful. A restricted form of $List(A)$ to $Vec(B)$, changing both the element type *and* the type constructor, poses fewer problems for meta-theory, since the possibilities for combination are much reduced. It is hoped that this benefit in the meta-theory may translate to benefits in the implementation, since adding such restrictions can help to control the more powerful coercion rules.

- A combination of techniques: there is no reason why the techniques mentioned here must be used in isolation.

Plastic currently provides simple measures to prevent the most obvious problems. When a coercion is declared, the system first searches for a coercion of identical type in the existing coercions. If one is found, then the new coercion is rejected. We also reject coercions whose type is convertible with $K \to K$, ie. has the same kind in the domain and the range, since such coercions will never be needed.

4.1 Dynamic Checking

4.1.1 The central idea

The coherence checking in [Bai98, Saï99] is based on their underlying graph data structure. This graph contains all coercions declared, and they test coherence over all known coercions by checking equivalence of all duplicate paths in the graph. But, checking *every* coercion isn't strictly required since it may never be used, and with a wider notion of coercion, it is impossible. Intuitively, we need only check coherence of the *actual coercions used*. This simplifies the problem, mainly because now we only check concrete coercions, rather than the (uncheckable) parametrized coercions. As well as efficiency gains where checking can be automated, there is a reduction in the burden of proof when coherence must be checked externally.

This restricted checking can be achieved by building up a set of concrete coercions whilst we are synthesising the type of a subterm. This is in contrast to building a *global* graph, as in Lego and Coq. The synthesis algorithm thus returns a type and a term with coercions already inserted, plus an additional structure representing the coercions used in the subterm. We may still need to store potential coercions (ie, parametrized coercions, coercion rules, basic coercions) in a graph-like structure, to support efficient calculation of a coercion term when a concrete coercion is required.

4.1.2 Technical details

The set of concrete coercions used in a subterm can be viewed as a set of graphs; a new coercion can be added to a single graph when its domain is the range of another coercion, or when its range is the domain of another coercion. Two graphs can be joined via a new coercion when its domain is the range of a coercion in one graph, and its range is the domain of a coercion in the second. Coherence problems arise when a duplicate path appears in the graph; notice that by adding a single coercion, we will have at most one duplicated path. The precise action required when a duplicate path is found will be considered below, as an implementation detail.

Next, we must consider how the coercion structure is manipulated as type synthesis proceeds. A variety of approaches seem possible. Firstly, the coercion structures can be merged compositionally, eg under an application node we synthesise the type of the subterms, to get two independent structures which must be merged in some way. This is likely to be algorithmically straight-forward, joining graphs via convertible nodes, although it could lead to many expensive conversion tests. The second option is to propagate the coercion structure in a depth-first fashion, ie the structure resulting from synthesis in the left sub-term is the starting point of synthesis in the right subterm. Merging is not required; we are back to the situation of adding a coercion at a time. We will choose between these approaches after studying their costs and benefits when applied to real examples.

Binders do present a definite problem. Firstly, what should the structure look like when it is moved outside the scope of a binder, when the bound variable occurs in the coercion terms or their types? A possible solution is to abstract the relevant terms, although this might be equivalent to re-introducing the parametrization of coercions which we wanted to avoid. Secondly, a conceptual difficulty: is an instance of a coercion under one bound variable comparable to an instance bound under a different variable? There does not seem to be an easy answer for this.

We may solve some of the problems by explicitly substituting for bound variables in the coercion structure when the relevant functional operation is applied to a term. For example, where $(t : T)_G$ denotes a term t with synthesised type T and coercion structure G, and $[x : K]G$ represents an abstraction of x over G, this means:

$$\frac{\Gamma \vdash (f : (x : K)K')_{[x:K]G_1} \quad \Gamma \vdash (t{:}K)_{G_2}}{\Gamma \vdash (f(t) : [t/x]K')_{(G_1(t)) \uplus G_2}}$$

4.1.3 Dynamic checking in practice

How will a system that implements dynamic checking behave? What happens when incoherencies are detected? And more importantly, how can a user remedy the situation? Type checking of a term will calculate the coercions necessary for it to be accepted; as each new coercion is added, a check will be made that (local) coherence is not broken. We envisage that synthesis will halt and report an error as soon as coherence is compromised. It may be possible to report all coherence problems in the whole term, but the simpler version will be easier to use. The question is now: what can the user do? There are several cases.

Firstly, coherence is maintained in theory but the proof cannot be generated automatically. Here, the user can either supply an explicit proof of this fact (ie, proving it within the system, eg by induction), or else force the implementation to assume such a proof, with the understanding that the correctness of the result is dependent upon such assumptions. The user can of course guarantee these assumptions meta-theoretically.

Secondly, declared coercions and "coercion templates" (ie the various forms of parametrized coercion) do give rise to real incoherence. That is, one coercion is in direct conflict with another, and no proof can make this problem disappear. One solution is to check whether the conflicting coercions represent what the user really meant; if not, then they should be removed from the system or replaced with the correct version. Another solution is to avoid relying on coercions in that particular case, ie to use explicit conversions or some form of type casting, to make the user's intentions clear.

4.1.4 Implementation details

This style of checking can be supported without great change to the current algorithms in Plastic. The main difference will be that type synthesis now returns, in addition to a term with explicit coercions and its type, a structure representing the coercions used in that term. We will implement a prototype of the ideas above, to enable experimentation and analysis in non-trivial examples; this will help the theoretical work and increase understanding of the practical implications.

There are several issues for a usable implementation which do not occur in the theoretical presentation. For example, if a user declares a coercion on his algebraic structures which is intensionally equal to a system default (eg both defined via π_1), then should the implementation show that the system default or the user's named coercion has been used? The second may be expected if the types being operated on are precisely those of the user's algebraic structure. We can aim for an idea of specificity of coercion, so that the most specific coercion is inserted; we will not discuss this further.

5 Conclusions and Future Work

The implementation of Coercive Subtyping in Plastic is work in progress. In addition to work on coherence, future work will include a systematic approach to coercion rules and to coercions which can be composed with themselves. Such issues are more than implementation problems, so further theory is required. We will also undertake larger examples of coercion use.

The notion of dynamic checking may help with practical implementations of coherence checking, but there is still much research to be done, both in the theory and in the pragmatic aspects of implementing it in a usable form. We plan to implement the basic infrastructure soon, so we can better understand the different kind of information available in non-trivial cases of coercion use, and to support the related theoretical investigations. The fact that coherence checking is a hard problem will be difficult to escape. Preliminary work is also pointing to another problem which may be equally difficult in practice. All coherence checking depends on some form of matching of type expressions. How to efficiently match types, and without imposing too many restrictions, must also be researched. For example, an expression E could reduce after a large computation to a concrete type T, but we don't want to invoke such computations each time we try a coercion.

It is possible that success will only come by detailed understanding of the various tradeoffs between logical power and implementability. For example, syntactic name matching (ie, the systems of Bailey and Saïbi) should probably be implemented as a subset, with good performance for many standard coercions. More powerful coercions could be supported, but the user may have to accept the performance implications.

There are interesting links between the problems of coercive subtyping and those of Qualified Types [Jon93]. Qualified types are an extension of the Hindley-Milner type system, and are the logical basis for recent extensions to Haskell [Has], such as type classes, lightweight extensible records, and multiple parameter classes. Issues of coherence and decidability do appear in implementations of these extensions. Some of the strategies being used on them, e.g. GHC's approach to multiple parameter classes [Pey] may be relevant and useful for coercive subtyping. We intend to study this further, in conjunction with our work with coercions in Haskell [LCK00].

Acknowledgements

The author thanks members of the Computer Assisted Reasoning Group (CARG) at the University of Durham for helpful discussions and comments, in particular Randy Pollack, for discussions on use of coercions in Coq. Thanks also to the anonymous referees, for their suggestions about improvements to the presentation.

References

[Asp] David Aspinall. Proof General WWW Site. `http://www.dcs.ed.ac.uk/home/proofgen`.

[Bai96] A. Bailey. Lego with implicit coercions. 1996. Draft.

[Bai98] A. Bailey. *The Machine-checked Literate Formalisation of Algebra in Type Theory.* PhD thesis, University of Manchester, 1998.

[BCGS91] V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance and explicit coercion. *Information and Computation*, 93, 1991.

[Bet97] Gustavo Betarte. Dependent record types, subtyping and proof reutilization. In *Proceedings of Workshop on Subtyping, inheritance and modular development of proofs [LS97]*, August 1997. See `http://-www.dur.ac.uk/CARG/cs-workshop97/`.

[BF99] G. Barthe and M.J. Frade. Constructor subtyping. In D. Swiestra, editor, *Proceedings of ESOP'99*, volume 1576 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 1999.

[Car88] L. Cardelli. Type-checking dependent types and subtypes. *Lecture Notes in Computer Science*, 306, 1988.

[CC99] C. Coquand and Th. Coquand. Structured type theory. In *Proc. Workshop on Logical Frameworks and Meta-languages (LFM'99)*, 1999.

[CG97] A. Compagnoni and H. Goguen. Typed operational semantics for higher order subtyping. Report ECS-LFCS-97-361, Dept. of Computer Science, University of Edinburgh, July 1997.

[CL98] P. Callaghan and Z. Luo. Mathematical vernacular in type theory based proof assistants. *User Interfaces for Theorem Provers (UITP'98), Eindhoven*, 1998.

[CL00a] P. Callaghan and Z. Luo. Implementation Techniques for Inductive Types. (submitted to Proceedings of TYPES'99), 2000.

[CL00b] Paul Callaghan and Zhaohui Luo. Plastic: an Implementation of Typed LF with Coercive Subtyping & Universes. (submitted to Journal of Automated Reasoning, special issue o n Logical Frameworks), 2000.

[Coq] Coq. Coq WWW Page. `http://pauillac.inria.fr/coq`.

[Coq91] Th. Coquand. An algorithm for testing conversion in Type Theory. In G. Huet and G. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.

[Gog95] H. Goguen. The metatheory of UTT. *Types for proofs and programs, LNCS 996*, 1995.

[Has] Haskell. Haskell WWW Site. `http://www.haskell.org`.

[Hue89] Gérard Huet. The constructive engine. In R. Narasimhan, editor, *A Perspective in Theoretical Computer Science*. World Scientific Publishing, 1989. Commemorative Volume for Gift Siromoney.

[JLS98] A. Jones, Z. Luo, and S. Soloviev. Some proof-theoretic and algorithmic aspects of coercive subtyping. *Types for proofs and programs (eds, E. Gimenez and C. Paulin-Mohring), Proc. of the Inter. Conf. TYPES'96, LNCS 1512*, 1998.

[Jon93] Mark Jones. Coherence for qualified types. Research report YALEU/DCS/RR-989, Yale University, September 1993.

[KLM99] R. Kiessling, Z. Luo, and J. M. McKinna. Some issues in functional programming with dependent types. Talk in the Annual Conference of TYPES'99, June 1999.

[LC98a] Z. Luo and P. Callaghan. Coercive subtyping and lexical semantics (extended abstract). *LACL'98*, 1998.

[LC98b] Z. Luo and P. Callaghan. Mathematical vernacular and conceptual wellformedness in mathematical language. *Proceedings of the 2nd Inter. Conf. on Logical Aspects of Computational Linguistics, LNCS/LNAI 1582*, 1998.

[LCK00] Zhaohui Luo, Paul Callaghan, and Robert Kiessling. Coercive subtyping in haskell. (draft), 2000.

[LMS95] G. Longo, K. Milsted, and S. Soloviev. A logic of subtyping. In *Proc. of LICS'95*, 1995.

[LP92] Z. Luo and R. Pollack. LEGO Proof Development System: User's Manual. LFCS Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh, 1992.

[LS97] Z. Luo and S. Soloviev, editors. *Proc of TYPES Working Group Workshop on Subtyping, Inheritance, and Modularisation of Proofs*, Durham, 1997.

[LS99a] Z. Luo and S. Soloviev. Dependent coercions. *The 8th Inter. Conf. on Category Theory and Computer Science (CTCS'99), Edinburgh, Scotland. Electronic Notes in Theoretical Computer Science*, 29, 1999.

[LS99b] Z. Luo and S. Soloviev. Dependent coercions. *The 8th Inter. Conf. on Category Theory and Computer Science (CTCS'99), Edinburgh, Scotland. Electronic Notes in Theoretical Computer Science*, 29, 1999. Revised version submitted to Theoretical Computer Science.

[Luo94] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, 1994.

[Luo97] Z. Luo. Coercive subtyping in type theory. *Proc. of CSL'96, the 1996 Annual Conference of the European Association for Computer Science Logic, Utrecht. LNCS 1258*, 1997.

[Luo99] Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130, 1999.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[NPS90] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: An Introduction.* Oxford University Press, 1990.

[P+a] S. L Peyton Jones et al. GHC Haskell Compiler WWW Site. `http://-www.haskell.org/ghc`.

[P+b] R. Pollack et al. Lego WWW Page. `http://www.dcs.ed.ac.uk/home-/lego`.

[Pey] S. Peyton Jones. Multi-parameter type classes in ghc. `http://research.microsoft.com/~simonpj/Haskell/multi-param.html`.

[Saï99] A. Saïbi. *Outils Génériques de Modélisation et de Démonstration pour la Formalisation des Mathématiques en Théorie des Types. Application à la Théorie des Catégories.* PhD thesis, INRIA, Rocquencourt, 1999.

[SL98a] S. Soloviev and Z. Luo. Coercion completion and conservativity in coercive subtyping. Talk given at TYPES'98, 1998.

[SL98b] S. Soloviev and Z. Luo. Coercive subtyping: coherence and conservativity. Draft paper, 1998.

[Tas97] A. Tasistro. *Substitution, record types and subtyping in type theory.* PhD thesis, Chalmers University of Technology, 1997.