Are subsets necessary in Martin-Löf type theory?

Simon Thompson

Computing Laboratory, University of Kent at Canterbury
Canterbury, CT2 7NF, U.K.
e-mail: sjt@ukc.ac.uk

Introduction

Martin-Löf's theory of types, expounded in [5, 6] and discussed at greater length in [8, 14] is a theory of types and functions or alternatively of propositions and proofs which has attracted much recent interest in the computing science community. There seems to have emerged a consensus that the system provides a good *foundation* for integrated program development and proof, but that for the system to be usable in practical projects a number of additions need to be made to it. Pre-eminent among these is the proposal to add a *subset* construction to the system, so that members of the type

$$\{ \, x : A \mid B \, \}$$

are those members of A with the property B. This is in contrast to the representation of such a type by

$$(\exists x : A) \,.\, B$$

whose members consist of pairs (a, p) with $a : A$ and $p : B[a/x]$ a *proof* of or *witness* to the fact that the property B is true of a. This latter representation is faithful to the principle of *complete presentation* which requires that it should be evident from any object that it has the type asserted of it. The witness is a proof of this fact.

In this paper after presenting an overview of the two important subset constructions we examine the reasons given for the addition of the subset type and argue that we can achieve the desired results *without* complicating the system by such an augmentation.

The first reason for adding a subset type is that it allows for the separation of the computational information in an object from the proof theoretic information it might contain – this we examine in sections 3 and 4 where we argue that this separation is better achieved by *naming* the appropriate portions of an object, using the axiom of choice where necessary to identify these portions.

Note that we take the axiom of choice as valid – this is the case for all Martin-Löf's systems, and is a simple consequence of the *strong* elimination rule for the existential quantifier, which allows the second projection from a pair to have dependent type. We should also observe that for most constructivists the axiom of choice is unexceptionable – given the interpretation of the quantifiers, a choice function can be read off in the obvious way.

The other reason advanced for the subset type is that it can contribute to the efficiency of evaluation, since by suppressing the proof-theoretic portion of an expression, any evaluation in this portion will no longer be necessary. We argue in section 5 that exactly the same effect is achieved if we use *lazy* evaluation to implement the system.

It is therefore evident that we can achieve the effects required *without* adding to and therefore complicating the system of type theory, as expounded in Martin-Löf's original papers, as long as we are prepared to work in a lazy implementation of the theory. Lazy implementations used to have the reputation of being slow, but recent work (see, for example, [10]) has shown that this need not be the case.

1 Type theory

In this section we provide a short review of those aspects of constructive type theory relevant to the discussion which follows.

The basic intuition underlying the system of type theory is that

to prove is to construct.

In particular,

- a proof of $A \wedge B$ is a pair of proofs of A and B;
- a proof of $A \Rightarrow B$ is a transformation of proofs of A into proofs of B;
- a proof of $A \vee B$ is either a proof of A or a proof of B;
- a proof of $(\forall x : A) \,.\, B(x)$ takes a in A to a proof of $B(a)$, and
- a proof of $(\exists x : A) \,.\, B(x)$ is a witness a in A together with a proof of $B(a)$.

This intuitive presentation can be formalised in a collection of deduction rules which mention the *judgement*

$$a : A$$

which is thought of as expressing

a is a proof of the formula A

Four rules are presented for each connective. A *formation* rule describes how the formula is formed, it is a rule of *syntax* in other words

Formation Rule for $\wedge$

$$\frac{A \ is \ a \ formula \quad B \ is \ a \ formula}{(A \wedge B) \ is \ a \ formula}(\wedge F)$$

A second rule gives circumstances under which a proof of the formula can be found – this is the *introduction* rule. In the case of conjunction, a proof can be formed from proofs of the component formulas.

Introduction Rule for $\wedge$

$$\frac{p : A \quad q : B}{(p, q) : (A \wedge B)} (\wedge I)$$

The *elimination* rule or rules embody the fact that proofs can only be constructed according to the introduction rule(s): any proof of a conjunction can be decomposed to yield proofs of the individual components.

Elimination Rules for $\wedge$

$$\frac{r : (A \wedge B)}{\mathit{fst}\ r : A} (\wedge E_1) \qquad \frac{r : (A \wedge B)}{\mathit{snd}\ r : B} (\wedge E_2)$$

Moreover, if we form a proof (a, b) of $A \wedge B$ from proofs of A and B and extract the proofs of A and B from (a, b) using $(\wedge E)$ we derive the proofs we started with – this is described by the *computation* rules.

Computation Rules for $\wedge$

$$\mathit{fst}\ (p, q) \ \rightarrow \ p \qquad \mathit{snd}\ (p, q) \ \rightarrow \ q$$

It is striking that these rules can also be thought of as rules for a typed functional programming language, if we replace '$\ldots$ *is a formula*' by '$\ldots$ *is a type*'. The rules for the conjunction are those for the product type.

In a similar way, implication can be thought of as forming the *function space*, disjunction a *sum* type and the absurd proposition an *empty* type. Rules for the function type are given in Figure 1. The connective $\Rightarrow$ is introduced by means of a λ-abstraction: the assumption of the object x of A is discharged in the process of forming $(\lambda x : A) . e$, as the variable x has become *bound*. The discharge is indicated by the surrounding brackets $[\ldots]$.

$$\frac{A \ \mathit{is\ a\ type} \quad B \ \mathit{is\ a\ type}}{(A \Rightarrow B) \ \mathit{is\ a\ type}} (\Rightarrow F) \qquad \frac{\begin{array}{c}[x : A] \\ \vdots \\ e : B\end{array}}{(\lambda x : A) . e \ : \ (A \Rightarrow B)} (\Rightarrow I)$$

$$\frac{q : (A \Rightarrow B) \quad a : A}{(q\ a) \ : \ B} (\Rightarrow E) \qquad ((\lambda x : A) . e)\ a \ \rightarrow \ e[a/x]$$

Fig. 1. Rules for function space

The formal system of type theory contains as well as the propositional connectives we have discussed, both basic types such as the natural numbers and lists and the *quantifiers* – we turn to these now. Further details of the full rules for systems of type theory can be found in [14] and elsewhere.

The universal quantifier can be thought of as defining a *generalised* function space in which the type of the result of a function depends upon the value of the argument(s).

Formation Rule for ∀

$$\frac{A\ \mathit{is\ a\ formula} \quad \begin{array}{c}[x:A]\\ \vdots\\ P\ \mathit{is\ a\ formula}\end{array}}{(\forall x:A)\,.\,P\ \mathit{is\ a\ formula}}(\forall F)$$

The dependence can be seen here from the fact that P can contain the variable x free, and so depend upon a value of type A. The universal quantifier is introduced by a λ-abstraction, where it is assumed that x occurs free in no other assumption than $x:A$.

Introduction Rule for ∀

$$\frac{\begin{array}{c}[x:A]\\ \vdots\\ p:P\end{array}}{(\lambda x:A)\,.\,p \ :\ (\forall x:A)\,.\,P}(\forall I)$$

The elimination and computation rules generalise those for implication.

Elimination Rule for ∀

$$\frac{a:A \quad f:(\forall x:A)\,.\,P}{f\ a \ :\ P[a/x]}(\forall E)$$

Computation Rule for ∀

$$((\lambda x:A)\,.\,p)\ a \quad \rightarrow \quad p[a/x]$$

A constructive interpretation of the *existential* quantifier takes objects of existential type to be *pairs*, the first half of which gives the witnessing element, and the second half the *proof* that this element has the property required.

Formation Rule for ∃

$$\frac{A\ \mathit{is\ a\ formula} \quad \begin{array}{c}[x:A]\\ \vdots\\ P\ \mathit{is\ a\ formula}\end{array}}{(\exists x:A)\,.\,P\ \mathit{is\ a\ formula}}(\exists F)$$

Introduction Rule for ∃

$$\frac{a:A \quad p:P[a/x]}{(a,p)\ :\ (\exists x:A)\,.\,P}(\exists I)$$

The elimination and computation rules show the decomposition of an existential proof object into its component parts.

Elimination Rules for $\exists$

$$\frac{p\ :\ (\exists x\!:\!A)\,.\,P}{Fst\ p\ :\ A}(\exists E_1') \qquad \frac{p\ :\ (\exists x\!:\!A)\,.\,P}{Snd\ p\ :\ P[Fst\ p/x]}(\exists E_2')$$

Computation Rules for $\exists$

$$Fst\ (p,q)\ \rightarrow\ p \qquad Snd\ (p,q)\ \rightarrow\ q$$

The existential quantifier can be thought of as a type constructor in a number of different ways. It forms an infinitary sum of the types $B(a)$ as a varies over the tag type A; it can be thought of as forming modules, when the type A is a universe, and most importantly here, it forms a *subset* of A, consisting of those elements of A with the property B. In keeping with a constructivist approach, the element a carries with it the proof that it belongs to the subset – otherwise how can it be said to reside there?

The elimination rule $(\exists E_2')$ is unusual in that its conclusion contains the proof object p on the right-hand side of a judgement; this is in contrast to the other rules of the system. These other rules reduce to the rules of first-order intuitionistic predicate calculus if the proof objects are omitted; this cannot be done with $(\exists E_2')$ since the proof object appears in the formula part of the judgement. The rules presented are equivalent to the *strong* elimination rule:

$$\frac{p\!:\!(\exists x\!:\!A)\,.\,B \quad \begin{array}{c}[x\!:\!A;\, y\!:\!B]\\ \vdots \\ c\!:\!C[(x,y)/z]\end{array}}{Cases_{x,y}\ p\ c\ :\ C[p/z]}(\exists E)$$

In turn, this rule has been shown in [13] to be equivalent to the axiom of choice plus a weaker rule of elimination which corresponds to the usual rule of elimination in first-order logic. We show the derivability of the axiom of choice from the strong rule now.

The axiom of choice has the statement

$$(\forall x\!:\!A)\,.\,(\exists y\!:\!B)\,.\,C(x,y) \Rightarrow (\exists g\!:\!A \Rightarrow B)\,.\,(\forall x\!:\!A)\,.\,C(x,(g\ x))$$

Suppose that $f\ :\ (\forall x\!:\!A)\,.\,(\exists y\!:\!B)\,.\,C(x,y)$ then

$$Fst\ (f\ x)\ :\ B$$

and

$$Snd\ (f\ x)\ :\ C(x, Fst(f\ x))$$

Therefore

$$\lambda x_A\,.\,(Fst\ (f\ x))\ :\ (A \Rightarrow B)$$

and we write g for this function. Also,

$$\lambda x_A \,.\, (Snd\,(f\;x)) \;:\; (\forall x\!:\!A)\,.\,C(x, Fst(f\;x))$$

giving

$$\lambda x_A \,.\, (Snd\,(f\;x)) \;:\; (\forall x\!:\!A)\,.\,C(x, (g\;x))$$

We thus have an object

$$(\lambda x_A \,.\, (Fst\;(f\;x))\;,\; \lambda x_A \,.\, (Snd\;(f\;x)))$$

of type

$$(\exists g\!:\!A \Rightarrow B)\,.\,(\forall x\!:\!A)\,.\,C(x, (g\;x))$$

Abstracting over f gives the proof of the axiom of choice.

2 The subset type

What are the formal rules for the subset type? The rules we give now were first proposed in [7], and used in [3], page 167, and [2], section 3.4.2. Formation is completely straightforward

Formation Rule for *Set*

$$\frac{A\ is\ a\ type \quad \begin{array}{c}[x\!:\!A]\\ \vdots \\ B\ is\ a\ type\end{array}}{\{\, x\!:\!A \mid B \,\}\ is\ a\ type}(SetF)$$

as is the introduction rule,

Introduction Rule for *Set*

$$\frac{a\!:\!A \quad p\!:\!B[a/x]}{a \;:\; \{\, x\!:\!A \mid B \,\}}(SetI)$$

How should a set be eliminated? If we know that $a\!:\!\{\, x\!:\!A \mid B \,\}$ then we certainly know that $a : A$, but also that $B[a/x]$. What we don't have is a specific proof that $B[a/x]$, so how could we encapsulate this? We can modify the existential elimination rule $(\exists E)$ so that the hypothetical judgement $c \;:\; C$ is derived assuming some $y\!:\!B[a/x]$, but that c and C cannot depend upon this y. We use the fact that $B[a/x]$ is provable, but we cannot depend on the proof y itself:

Elimination Rule for *Set*

$$\frac{a\!:\!\{\, x\!:\!A \mid B \,\} \quad \begin{array}{c}[x\!:\!A; y\!:\!B]\\ \vdots \\ c(x)\!:\!C(x)\end{array}}{c(a) \;:\; C(a)}(SetE)$$

where y is not free in c or C. Since no new operator is added by the elimination rule, there is no computation rule for the subset type. We should note that this makes these rules different from others in type theory. This is also evident from the fact that they fail to satisfy the *inversion principle* of [12]

[11] shows that these rules are weaker than might at first be thought, especially if we adopt an *intensional* version of type theory. In fact, we cannot in a consistent manner derive the formula

$$(\forall x : \{ z : A \mid P(z) \}) \,.\, P(x) \tag{1}$$

for most formulas P. This has the consequence that we cannot derive functions to take the head and tail of a non-empty list, if we choose to represent the type of non-empty lists by a subset type,

$$\{ l : [A] \mid nonempty\, l \}$$

where the predicate *nonempty* is defined by a recursion over a universe thus:

$$\begin{array}{lll} nonempty\ [\,] & \equiv_{df} & \bot \\ nonempty\ (a :: x) & \equiv_{df} & \top \end{array}$$

The situation in the extensional theory is better, but there are still cases of the formula (1) which are not derivable consistently. Because of these weaknesses, Martin-Löf proposed a new *subset* theory which incorporates the judgement P *is true* into the system.

If the representation of the judgement is to be an improvement on TT, as far as subsets are concerned, it is desirable that the system validates the rule

$$\frac{a \ :\ \{ x : A \mid P \}}{P(a)\ is\ true} \tag{2}$$

This can be done if we move to a system in which propositions and types are distinct. In [8] can be found the **subset theory** in which the new judgements

$$P\ prop \qquad \text{and} \qquad P\ is\ true$$

are added to the system, together with a set of *logical* connectives, distinct from the type forming operations introduced in their extensional version of type theory. This system does allow the derivation of (2) but at the cost of losing the isomorphism between propositions and types and making the system more complex.

3 What is a specification?

The judgement $a : A$ can be thought of as expressing 'a proves the proposition A' and 'a is an object of type A', but it has also been proposed, in [6, 9] for example, that it be read as saying

a is a program which meets the specification A (†)

It is misleading to apply this interpretation to every judgement $a : A$. Take for instance the case of a function f which sorts lists; this has type $[A] \Rightarrow [A]$, and so,

$$f \;:\; [A] \Rightarrow [A]$$

Should we therefore say that it meets the specification $[A] \Rightarrow [A]$? It does, but then so do the identity and the reverse functions! The type of a function is but one aspect of its specification, which should describe the relation between its input and output. This characterisation takes the form

> The result $(f\ l)$ is ordered and a permutation of the list l

for which we will write $S(f)$. To assert that the specification can be met by some implementation, we write

$$(\exists f : [A] \Rightarrow [A]) \,.\, S(f)$$

What form do objects of this type take? They are pairs (f, p) with $f : [A] \Rightarrow [A]$ and p a proof that f has the property $S(f)$. The confusion in (†) is thus that the object a consists not of a program meeting the specification, but of such a program together with a *proof* that it meets that specification.

In the light of the discussion above, it seems sensible to suggest that we conceive of specifications as statements $(\exists o : T) \,.\, P$, and that the formal assertion

$$(o, p) \;:\; (\exists o : T) \,.\, P$$

be interpreted as saying

> The object o, of type T, is shown to meet the specification P by the proof object p.

an interpretation which combines the logical and programming interpretations of the language in an elegant way. This would be obvious to a constructivist, who would argue that we can only assert (†) if we have the appropriate evidence, namely the proof object.

In developing a proof of the formula $(\exists o : T) \,.\, P$ we construct a pair consisting of an object of type T and a proof that the object has the property P. Such a pair keeps separate the computational and logical aspects of the development, so that we can extract directly the computational part simply by choosing the first element of the pair.

There is a variation on this theme, mentioned in [8] for instance, which suggests that a specification of a function should be of the form

$$(\forall x : A) \,.\, (\exists y : B) \,.\, P(x, y) \tag{3}$$

Elements of this type are functions F so that for all $x : A$,

$$F\ x \;:\; (\exists y : B) \,.\, P(x, y)$$

and *each* of these values will be a pair (y_x, p_x) with

$$y_x : B \qquad \text{and} \qquad p_x : P(x, y)$$

The pair consists of value and proof information, showing that under this approach the program and its verification are inextricably mixed. It has been argued that the only way to achieve this separation is to replace the inner existential type with a *subset* type, which removes the proof information p_x. This can be done, but the intermingling can be avoided *without* augmenting the system – we simply have to give the intended function a *name*. That such a naming can be achieved in general is a simple consequence of the *axiom of choice*, which states that

$$(\forall x : A) . (\exists y : B) . P(x, y) \Rightarrow (\exists f : A \Rightarrow B) . (\forall x : A) . P(x, f\,x)$$

Applying *modus ponens* to this and (3) we deduce the specification

$$(\exists f : A \Rightarrow B) . (\forall x : A) . P(x, f\,x) \tag{4}$$

Note that the converse implication to that of the axiom of choice is easily derivable, making the two forms of the specification logically equivalent.

It is worth noting that some functions are not specified simply by their input/output relation, one example being a hashing function[1]. This means that specifications will necessarily have the $(\exists o : T) . P$ form in general.

This analysis of specifications makes it clear that when we seek a program to meet a specification, we look for the *first* component of a member of an existential type; the second proves that the program meets the constraint part of the specification. As long as we realise this, it seems irrelevant whether or not our system includes a type of first components, which is what the subset type consists of. There are other arguments for the introduction of a subset type, which we turn to now.

4 Subsets in specifications

We have seen that the intermingling of computation and verification which appears to result from an interpretation of specifications as propositions can be avoided by the expedient of using the axiom of choice in the obvious way.

In this section we look at other uses of the subset type within specifications and show that in many of these we can again avoid the subset type by separating from a complex specification exactly the part which is computationally relevant in some sense. This is to be done by *naming* in an appropriate manner the operations and objects sought, as we did in the previous section when we changed the $\forall\exists$ specification into an $\exists\forall$ form. This reversal of quantifiers which arises by naming the function is known to logicians as **Skolemizing** the quantifiers. We believe the alternative is superior for two reasons:

[1] I am grateful to Michael O'Donnell for this observation.

– it is a solution which requires no addition to the system of type theory, and
– it allows for more delicate distinctions between proof and computation.

The method of Skolemizing can be used in more complex situations, as we now see.

Take as an example a simplification of the specification of the Dutch (or Polish) national flag problem as given in [8]. We now show how it may be written without the subset type. The original specification has the form

$$(\forall x : A) \,.\, \{\, y : \{\, y' : B \mid C(y') \,\} \mid P(x, y) \,\}$$

with the intention that for each a we find b in the subset $\{\, y' : B \mid C(y') \,\}$ of B with the property $P(a, b)$. If we replace the subsets by existential types, we have

$$(\forall x : A) \,.\, (\exists y : (\exists y' : B) \,.\, C(y')) \,.\, P(x, y)$$

This is logically equivalent to

$$(\forall x : A) \,.\, (\exists y : B) \,.\, (\, C(y) \wedge P(x, y) \,) \tag{5}$$

and by the axiom of choice to

$$(\exists f : A \Rightarrow B) \,.\, (\forall x : A) \,.\, (\, C(f\ x) \wedge P(x, (f\ x)) \,)$$

which is inhabited by functions *together with proofs of their correctness*. It can be argued that this expresses in a clear way what was rather more implicit in the specification based on sets – the formation of an existential type bundles together data and proof, the transformation to (5) makes explicit the unbundling process.

As a second example, consider a problem in which we are asked to produce for each a in A with the property $D(a)$ some b with the property $P(a, b)$. There is an important question of whether the b depends just upon the a, or upon both the a and the proof that it has the property $D(a)$. In the latter case we could write the specification thus:

$$(\forall x : (\exists x' : A) \,.\, D(x')) \,.\, (\exists y : B) \,.\, P(x, y)$$

and Skolemize to give

$$(\exists f : (\exists x' : A) \,.\, D(x') \Rightarrow B) \,.\, (\forall x : (\exists x' : A) \,.\, D(x')) \,.\, P(x, (f\ x))$$

If we use the equivalence between the types

$$((\exists x : X) \,.\, P) \Rightarrow Q \qquad\qquad (\forall x : X) \,.\, (P \Rightarrow Q)$$

(which is the logical version of the isomorphism between 'curried' and 'uncurried' versions of binary functions) we have

$$(\exists f : (\forall z : A) \,.\, (D(z) \Rightarrow B)) \,.\, (\forall x' : A) \,.\, (\forall p : D(x')) \,.\, P((x', p), (f\ x'\ p))$$

which makes manifest the functional dependence required. Observe that we could indeed have written this formal specification directly on the basis of the informal version from which we started.

If we do *not* wish the object sought to depend upon the proof of the property D, we can write the following specification:

$$(\exists f : A \Rightarrow B) \,.\, (\forall x' : A) \,.\, (\forall p : D(x')) \,.\, P((x', p), (f\ x')) \tag{6}$$

in which it is plain that the object $(f\ x')$ in B is not dependent on the proof object $p : D(x')$. Observe that there *is* still dependence of the property P on the proof p; if we were to use a subset type to express the specification, thus, we would have something of the form

$$(\forall x' : \{\, x' : A \mid D(x') \,\}) \,.\, (\exists y : B) \,.\, P'(x', y)$$

where the property $P'(x, y)$ relates $x' : A$ and $y : B$. This is equivalent to the specification

$$(\exists f : A \Rightarrow B) \,.\, (\forall x' : A) \,.\, (\forall p : D(x')) \,.\, P'(x', (f\ x'))$$

in which the property P' must not mention the proof object p, so that with our more explicit approach we have been able to express the specification (6) which cannot be expressed under the naïve subset discipline.

5 Computational Irrelevance; Lazy Evaluation

The natural definition of the 'head' function on lists is over the type of non-empty lists, given thus:

$$(nelist\ A) \equiv_{df} (\exists l : [A]) \,.\, (nonempty\ l)$$

where the predicate *nonempty* was defined above. The head function itself, *hd*, is given by

$$\begin{array}{lll} hd & : & (nelist\ A) \Rightarrow A \\ hd\ ([\,], p) & \equiv_{df} & abort_A\ p \\ hd\ ((a :: x), p) & \equiv_{df} & a \end{array}$$

which is formalised in type theory by a primitive recursion over the list component of the pair.

Given an application

$$hd\ ((2 :: \ldots), \ldots)$$

computation of the result to 2 can proceed in the absence of any information about the elided portions. In particular, the *proof* information is not necessary for the process of computation to proceed in such a case. Nonetheless, the proof information is crucial in showing that the application is properly typed; we cannot apply the function to a bare list, as that list might be empty. There is thus a tension between what are usually thought of as the *dynamic* and *static* parts of the language. In particular it has been argued that if no separation is achieved, then the efficiency of programs will be impaired by the welter of irrelevant information which they carry around – see section 3.4 of [2] and section 10.3 of [3].

Any conclusion about the efficiency of an object or program is predicated on the evaluation mechanism for the system under consideration, and we now argue that a *lazy* or outermost first strategy has the advantage of not evaluating the computationally irrelevant.

If we work in an intensional system of type theory, then using the results of [5] the system is both strongly normalising and has the Church Rosser property. This means that *every* sequence of reductions will lead us to the same result. Similar results are valid if we evaluate to weak head normal form in the extensional case.

We can therefore choose how expressions are to be evaluated. There are two obvious choices. **Strict** evaluation is the norm for imperative languages and many functional languages (Standard ML, [4], is an example). Under this discipline, in an application like

$$f\ a_1\ \ldots\ a_n$$

the arguments a_i are evaluated fully before the whole expression is evaluated. In such a situation, if an argument a_k is computationally irrelevant, then its evaluation will degrade the efficiency of the program. The alternative, of **normal order** evaluation is to begin evaluation of the whole expression, prior to argument evaluation: if the value of an argument is unnecessary, then it is not evaluated.

To be formal about the definition, we say that evaluation in which we always choose the leftmost outermost redex is **normal order** evaluation. If in addition we ensure that no redex is evaluated more than once we call the evaluation **lazy**. (For more details on evaluation strategies for functional languages, see [10], for example).

In a language with structured data such as pairs and lists, there is a further clause to the definition: when an argument is evaluated it need not be evaluated to normal form; it is only evaluated to the extent that is necessary for computation to proceed. This will usually imply that it is evaluated to weak head normal form (see [10]). This means that, for example, an argument of the product type $A \wedge B$ will be reduced to a pair (a, b), with the sub-expressions a and b as yet unevaluated. These may or may not be evaluated in subsequent computation.

Under lazy evaluation computationally irrelevant objects or components of structured objects will simply be *ignored*, and so no additional computational overhead is imposed. Indeed, it can be argued that the proper definition of computational relevance would be that which chose just that portion of an expression which is used in calculating a result under a lazy evaluation discipline.

Another example is given by the following example of the *quicksort* function over lists. Quicksort is defined by

$$qsort\ l \equiv_{df} qsort'\ l\ (\#l)\ p$$

where p is the canonical proof that $(\#l) \leq (\#l)$. The auxiliary function is given by

$$qsort' \ :\ (\forall n : N)\,.\,(\forall l : [N])\,.\,((\#l \leq n) \Rightarrow [N])$$

$$
\begin{array}{lll}
qsort'\ n\ [\,]\ p & \equiv_{df} & [\,] \\
qsort'\ 0\ (a :: x)\ p & \equiv_{df} & abort_{[N]}p' \\
qsort'\ (n+1)\ (a :: x)\ p & & \\
\quad \equiv_{df} \quad qsort'\ n\ (filter\ (lesseq\ a)\ x)\ p_1 & & \\
\qquad\qquad ++\ [a]\ ++ & & \\
\qquad\qquad qsort'\ n\ (filter\ (greater\ a)\ x)\ p_2 & &
\end{array}
$$

The function has three parameters: a list (l), a natural number (n), and a proof that the length of l is less that or equal to n. In the second clause of the definition we use the proof p to construct a proof p' that 0 is smaller than itself, a contradiction. In the recursive calls to the function, we construct proofs p_1 and p_2 which witness the facts that $(filter\ (lesseq\ a)\ x)$ and $(filter\ (greater\ a)\ x)$ have length at most n if x has.

We have built an implementation of a system of type theory without universes by means of a translation of it into Miranda which is implemented in a lazy fashion. The quicksort function above will sort a list without calculating any of the terms p_i in any of the invocations of the function – proof of their computational irrelevance.

There is one drawback to the lazy implementation – no irrelevant terms are evaluated, but there *are* cases in which tuples are formed and destroyed, as were the tuples in the quicksort example. It seems too high a price to pay for the programmer to have to include for her- or himself an indication of how a program may be optimised, especially as this kind of use analysis can be performed most effectively by the techniques of abstract interpretation, as discussed in [1] for instance. Linked to this is the syntactic characterisation of computational relevance, which involves an examination of the different forms that types (*i.e.* propositions) can take – to be found in section 3.4 of [2]. It is not hard to see that under lazy evaluation the objects deemed to be irrelevant will not contribute to the final result, and will remain unevaluated.

6 Conclusion

To summarise, there are two responses to the use of subsets in type theory. Their use in separating the computational from the proof theoretic can be achieved using the appropriate names for functions whose existence is assured by the validity of the axiom of choice in type theory.

If proof theoretic information remains in an expression, we contend that if it is indeed irrelevant to the computational behaviour of a function, it will not be evaluated under a lazy evaluation strategy, and so we advocate this as an implementation technique which avoids the unnecessary evaluation which is a consequence of a strict evaluation scheme. As we mentioned earlier, there will be some cases in which structures are formed needlessly – we see their elimination as the role of the implementation of the system, and would view abstract interpretation as an ideal tool for this purpose.

Using the subset type to represent a subset brings problems; as we saw in the previous section, it is not possible in general to recover the witnessing information from a subset type, especially in an intensional system like TT, and so in these cases, the existential type should be used, retaining the witnessing information. Even in cases where such information can be recovered, we gain this only at the cost of having to work in a more complex system, especially in the case where the addition of the judgement P *is true* will give a confusion between similar results in the type and proposition modes.

References

1. Samson Abramsky and Chris Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis-Horwood, 1987.
2. Roland Backhouse, Paul Chisholm, Grant Malcolm, and Erik Saaman. Do-it-yourself type theory. *Formal Aspects of Computing*, 1, 1989.
3. Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall Inc., 1986.
4. Robert Harper. Introduction to Standard ML. Technical Report ECS-LFCS-86-14, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, November 1986.
5. Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H. Rose and J. C. Shepherdson, editors, *Logic Colloquium 1973.* North-Holland, 1975.
6. Per Martin-Löf. Constructive mathematics and computer programming. In C. A. R. Hoare, editor, *Mathematical Logic and Programming Languages.* Prentice-Hall, 1985.
7. Bengt Nordström and Kent Petersson. Types and specifications. In *IFIP'83.* Elsevier, 1983.
8. Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory — An Introduction*, volume 7 of *International Series of Monographs on Computer Science.* Oxford University Press, 1990.
9. Kent Petersson and Jan Smith. Program derivation in type theory: The Polish flag problem. In Peter Dybjer et al., editors, *Proceedings of the Workshop on Specification and Derivation of Programs.* Programming Methodology Group, University of Goteborg and Chalmers University of Technology, 1985. Technical Report, number 18.
10. Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.
11. Anne Salvesen and Jan Smith. The strength of the subset type in Martin-Löf's type theory. In *Proceedings of the Third Annual Symposium on Logic in Computer Science.* IEEE Computer Society Press, 1989.
12. Peter Schroeder-Heister. Judgements of higher levels and standardized rules for logical constants in Martin-Löf's theory of logic. In Peter Dybjer et al., editors, *Proceedings of the Workshop on Programming Logic.* Programming Methodology Group, University of Goteborg and Chalmers University of Technology, 1989. Technical Report, number 54. This paper was written in 1985.
13. Marco D. G. Swaen. *Weak and Strong Sum-Elimination in Intuitionistic Type Theory.* PhD thesis, University of Amsterdam, 1989.
14. Simon J. Thompson. *Type Theory and Functional Programming.* Addison Wesley, 1991.

This article was processed using the LaTeX macro package with LLNCS style