

Deep Types for SML: Theory and Practice

Stephen Gilmore
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh
King's Buildings, Mayfield Road
Edinburgh EH9 3JZ, United Kingdom
Tel. +44 131 650 5189
Fax +44 131 667 7209
Stephen.Gilmore@ed.ac.uk

Laura Korte^{*}
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh
King's Buildings, Mayfield Road
Edinburgh EH9 3JZ, United Kingdom
Tel. +44 131 650 5252
Fax +44 131 667 7209
L.Korte@sms.ed.ac.uk

ABSTRACT

In this paper we will present a *deep type system* for Standard ML. Deep type systems aim to narrow the type information gap that arises when imperative features are added to a functional language. By narrowing this gap, they can aid the programmer to increase program safety. This increased safety is accomplished by means of specifying both an inference system for deep types and a way of imposing constraints on them. The deep type system was developed in analogy with the regular (or *shallow*) Standard ML type system, in which types are automatically derived by the type inference algorithm and constraints may be imposed by specifying signatures. In this paper we will present both the design and some practical applications of the deep type system for Standard ML.

Keywords

Program Analysis, Type and Effect Systems, Type Inference, Standard ML

1. INTRODUCTION

Among the population of functional languages, there are the pure and the impure, the lazy and the strict, the typed and the untyped. These days, every single combination out of these three attributes, each with its own advantages and disadvantages, has been implemented at least once. However, if we consider just the typed impure languages for a moment, we notice that it is only the functional part of these languages in which the attribute ‘typed’ has any real meaning. That is, the imperative components and their side effects do not leave any trace in the type of a program. Judging by just its type, we cannot tell if a function updates any references, what exceptions it might throw or what IO-actions it can

^{*}Corresponding author.

perform. Even though this simplification was crucial in the design of Standard ML’s type system (see [15]), we would like to give the programmer some additional control over his program by allowing him to not only specify constraints on its functional types, but also on its side effects.

In this paper we will present a suggestion of how to extend the shallow¹ type system of a typed impure functional language (Standard ML) to include these and other imperative features and their side effects, as first described in [6] and [7]. The resulting system will be a *deep type system*. Our research is closely related to the Type and Effect discipline as described in [10], [1] and [11], in which every function is assigned not only a type, but also an effect, which is a formal representation of a function’s side effects. The contribution we hope to make towards this discipline is the introduction of assertions, which declare additional information about types. We shall see that the assertions owe much of their expressiveness to the *ghost functions* and *variables* (see [2]), and that defining one or more assertions for a function can be seen as the deep type equivalent of specifying the function’s shallow type in a Standard ML signature.

In the next chapter we will formally introduce the deep type system and its components. After this formalisation, there will be a chapter on the applications of the deep type system, followed by a section on related work and the conclusions.

2. THEORY

In the past, Standard ML’s type inference system has proven to be a very useful tool in analysing the type of a function (see [3]). Furthermore, Standard ML’s constraint system allows the programmer to impose arbitrary constraints on these types, by (either fully or partially) specifying them in a signature.

As mentioned before, the deep type system will be developed by analogy with Standard ML’s shallow type system. Like the latter, it will consist of an inference system and a constraint system. Apart from one or two exceptions, the

¹We refer to Standard ML’s type system as shallow not because it is uninteresting, but because it does not expose details of the function implementation by “looking inside” functions, whereas our deep type system does.

<i>effect</i>	::=	<i>past</i> <i>future</i> <i>effect</i> + <i>effect</i>
<i>past</i>	::=	<i>past</i> @ <i>past</i> <i>past</i> + <i>past</i> <i>H</i> (<i>exn</i> , <i>past</i>) <i>atom</i> <i>v</i> <i>name</i>
<i>future</i>	::=	<i>lambda</i> <+ <i>future</i> > <i>name</i> ϵ
<i>lambda</i>	::=	$\lambda v : effect_1.effect_2$
<i>atom</i>	::=	<i>T</i> (<i>exn</i>) <i>U</i> (<i>ref</i>) <i>R</i> (<i>ref</i>) ϵ
<i>ref</i>	::=	<i>vid</i> <i>names</i> # <i>ref</i> <i>name.ref</i>
<i>names</i>	::=	set of <i>name</i>

Figure 1: Syntax of Deep Types

outline of the deep type inference system will not differ much from that of shallow types. The deep type constraint system however will have an entirely different format from the one for shallow types. This discrepancy has a purely practical reason from the programmer’s point of view: because deep types – by definition – have a very complicated structure and can quickly become complex, it would be too arduous for the programmer to specify constraints on them the way it is done for shallow types in Standard ML.

In contrast to the polymorphic type definitions which Standard ML provides for its shallow types, we will design an assertion system in which the programmer can specify some properties he would like a function to have. These properties will then be checked against the deep type of the function and the programmer will be informed of the results of this check. To improve the expressiveness of the property assertions, we will make use of ghost functions and variables.

2.1 The Deep Type Inference System

We will now describe the structure of the deep types and their operational semantics.

2.1.1 Deep Type Design

The side effects we would like our deep types to account for are raising and handling exceptions and accessing and updating references. An example of a Standard ML type system dealing with exceptions only can be found in [8], but this system does not consider the side effects caused by references. Furthermore, we would also like to know the order in which these side effects occur, and if all executions of a function necessarily exhibit a certain side effect, or if the side effect is merely a possibility. All of this information will be represented by an *effect* term as defined in Figure 1.

The grammar in Figure 1 shows that an effect is either a simple effect or a sum of effects. Starting with the first option, a simple effect, we note that it consists of two sep-

$$\begin{array}{l}
 \mathcal{S} \vdash \mathbf{fn} \ x \Rightarrow x \Rightarrow \alpha \& \epsilon_1 \parallel \lambda v : (\phi \parallel \psi). \epsilon_2 \parallel \psi \\
 \mathcal{S} \vdash 5 \Rightarrow \mathbf{int} \& \epsilon_3 \parallel \epsilon_4 \\
 \mathcal{S} = \mathcal{U}(\phi \parallel \psi, \epsilon_3 \parallel \epsilon_4) = \{\phi \mapsto \epsilon_3, \psi \mapsto \epsilon_4\} \\
 \hline
 \mathcal{S} \vdash (\mathbf{fn} \ x \Rightarrow x) \ 5 \Rightarrow \tau \& \epsilon_3 @ \epsilon_1 @ \epsilon_2 \parallel \psi
 \end{array}$$

- where \mathcal{U} is unification, yielding a set of constraints.
- and \mathcal{S} is set the constraints.

Figure 2: How lambdas in the deep type system work.

arate terms - *past* and *future* - which are paired by means of the || operator. The *past* term represents the side effects which the SML expression has exhibited up to this point, whereas the *future* term represents the potential behaviour of a function, which will only be exhibited when applied to an argument. Naturally, the *future* term of a constant will be empty, as will the future term of a fully instantiated function. For example, the deep type of the integer 6 will be $\epsilon \parallel \epsilon$ and even though the deep type of an ML function *fac* - which computes the factorial of a number - may have a complex future, its fully instantiated version *fac* 6, will not. Instead, its deep type will be of the form $\phi \parallel \epsilon$, where ϕ depends on the implementation of *fac*.

Note that a *future* term may assume three different forms: that of a lambda term, that of a variable, or ϵ . While the two latter are self-explanatory, the first one deserves a little more explanation. A lambda term in the deep type system is of the form $\lambda v : effect_1.effect_2$. The *effect*₁ term is an effect composed solely of variables and epsilons and has the purpose of binding its variables through unification. The variables will be bound to the concrete values in the deep type of argument *v* and can be used in the *effect*₂ term as well. The unification process takes place during functional application as can be seen in the example shown in Figure 2 and rule (*app*/8) of the operation semantics in Figure 6 to 14.

Next, a *past* term is built up from the *atom* terms for throwing an exception (*T*(*exn*)), updating a reference (*U*(*ref*)) and accessing the contents of a reference (*R*(*ref*)). The operators holding these *atom* terms together represent *sequence* (@), *choice* (+) and *exception handling* (*H*(*exn*, *past*)). This last one always occurs in a sum, together with another *past* term representing the result when an exception *exn* is caught. It can be seen as a *choice* between the first and the second *past* term, but unlike a regular *choice*, it also handles all instances of *exn* that may arise from within the first *past* term.

The grammar in Figure 1 shows that, besides a simple term consisting of a *past* and a *future* term, an effect may also take the form of a sum. A sum indicates non-determinism in the control flow and can be described as an *exclusive or* operator. For example, if a function *f* has deep type $\phi_1 + \phi_2 + \phi_3$, it will - when supplied with an argument *a* - exhibit the behaviour described by ϕ_1 , or the behaviour described by ϕ_2 , or the behaviour described by ϕ_3 , depending on the value of the argument *a*, and the state of other program variables which it uses.

```

val r1 = ref 0

fun f x =
  let val r2 = ref 5
      in x + !r2
      end

local val r3 = ref 0
in fun inc () = (r3 := !r3 + 1 ; !r3)
end

local fun g x =
  let val r4 = ref 0
      in x + !r4
      end
in val i = g 5
end

```

Figure 3: References in the deep type system.

The *ref* terms in the deep type system refer to Standard ML expressions of the reference type. In a Standard ML program, these expressions can either be defined at top level (*vid*), within a ‘let’-expression (*name.vid*), or within a ‘local’-expression (*names # vid*). Nested terms are also possible. Figure 3 shows examples of the references *r1*, *f.r2*, *{inc}#r3* and *{i}#g.r4*. Note that we have adapted the derived forms of Standard ML syntax like **fun** in all our example programs to make them more readable.

In order to give a formal description of the meaning of all these deep types, we will now continue with the operational semantics of the deep type inference system.

2.1.2 Operational Semantics

The operational semantics of the deep type inference system is based on that of Standard ML as defined in [9]. The numbers which appear in the name of each rule, refer to the numbers assigned to the rules in the *Static Semantics of the Core* part of [9]. You will not find a counterpart for every rule from [9] in this paper, because we are only defining the deep type system for a subset of Standard ML expressions. The syntax of this subset is shown in Figure 4. Figures 6 to 14 form the complete operational semantics of the deep type inference system for this subset.

Figure 13 and Figure 14 show the rules without number. These rules do not have a counterpart in [9] because they encapsulate the semantics of a specific function, rather than a general language feature. However, these rules are also almost solely responsible for the creation of the atomic deep types, hence our motivation to include them in the operational semantics of deep types.

We will now discuss some of the notations and constructs we will need in order to define the operational semantics, after which we will go over some of the most important rules in some more detail.

A context Γ in the deep type inference system is a four tuple (C, NE, N, S) , composed of a context C as defined in [9]; a (possibly empty) list of *Loc* and *Let* environments NE ;

```

atpat ::= -
      | scon
      | <op> longvid
      | ( pat )

pat ::= atpat

atexp ::= scon
      | <op> longvid
      | let dec in exp end
      | ( exp )

exp ::= atexp
     | exp atexp
     | exp1 vid exp2
     | exp handle match
     | raise exp
     | fn match

match ::= mrule < | match >

mrule ::= pat => exp

dec ::= val tyvarseq valbind
     | local dec1 in dec2 end
     | dec1 (;) dec2

valbind ::= pat = exp
        | rec valbind

```

Figure 4: Syntax of SML fragment

the name of the current environment N and a function S , which maps deep types to their substitutions (or set of constraints). The context C is present for obvious reasons. The *nested environment* NE allows the type inference system to keep track of its absolute location within the program, which it needs in order to lookup or specify references. The name of the current environment N – which can be either a single name (within a function body or *Let* environment), or a set of names – is necessary in order to specify the recursive environments. Finally, the function S holds all substitutions. In the operational semantics, it will be updated by rules which yield a substitution as a result of the unification of two deep types.

In order to simplify the operational semantics, we will introduce some meta notation for the frequently used operations. First of all, we will adapt the meta notation used in [9] like *projection* (VE of E), *injection* (VE in Env) and *modification* ($E + VE, C \oplus TE$). Second, we will overload the $::$ -operator as demonstrated by: $L :: \Gamma$, which will stand for $(C$ of $\Gamma, L :: NE$ of Γ, \mathcal{N} of Γ, S of $\Gamma)$. Third, we will introduce the \triangleleft -operator in $\Gamma \triangleleft N$, which will be meta notation for $(C$ of Γ, NE of Γ, N, S of $\Gamma)$.

The notational approach to unification used in the operational semantics differs slightly from the standard approach, in that it unifies pairs consisting of a deep type and its constraint set, instead of merely constraint sets. However, the first element of the pair, i.e. the deep type, can be specified as a set of constraints. Furthermore, because the disjoint union of this set and the deep type’s initial constraint set will yield one single constraint set, it follows that the two approaches are equivalent. The formal definition of the notion of unification \mathcal{U}' as used in the operational semantics, is printed below. Note that unification will always take place within a context Γ , that S denotes S of Γ and that \mathcal{U} is stan-

$$\begin{aligned}
& \text{Basic}((\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta) = \\
& \phi_1 \parallel \lambda v_1: (\phi_2 \parallel \lambda v_2: (\phi_3 \parallel \phi_4) \cdot \phi_5 \parallel \phi_6) \cdot \phi_7 \parallel \lambda v_3: (\phi_8 \parallel \phi_9) \cdot \phi_{10} \parallel \phi_{11} \\
& \text{Basic}(\text{int} \rightarrow \text{int} \rightarrow \text{int}) = \\
& \phi_1 \parallel \lambda v_1: (\phi_2 \parallel \epsilon) \cdot \phi_3 \parallel \lambda v_2: (\phi_4 \parallel \epsilon) \cdot \phi_5 \parallel \epsilon
\end{aligned}$$

Figure 5: Two examples of the *Basic* function.

ward unification, yielding a set of constraints. Furthermore, both in the definition of unification and in the operational semantics, we have used τ , τ' , etc. to range over shallow types and all of the other Greek lower case letters, except ϵ , to range over deep types.

$$\mathcal{U}'(\phi, \psi) = \mathcal{U}((\phi, s_1), (\psi, s_2)) \quad \text{where} \quad \begin{aligned} s_1 &= \mathcal{S}(\phi) \\ s_2 &= \mathcal{S}(\psi) \end{aligned}$$

As mentioned before, the unification process yields a set of constraints, which can be used for substitution. Substitution on deep types is defined as follows:

$$\begin{aligned}
(\phi_1 \parallel \phi_2)[\psi/v] &= (\phi_1[\psi/v]) \parallel (\phi_2[\psi/v]) \\
(\phi_1 @ \phi_2)[\psi/v] &= (\phi_1[\psi/v]) @ (\phi_2[\psi/v]) \\
(\phi_1 + \phi_2)[\psi/v] &= (\phi_1[\psi/v]) + (\phi_2[\psi/v]) \\
H(e, \phi_1, \phi_2)[\psi/v] &= H(e, \phi_1[\psi/v], \phi_2[\psi/v]) \\
T(e)[\psi/v] &= T(e) \\
U(r)[\psi/v] &= U(r) \\
R(r)[\psi/v] &= R(r) \\
\epsilon[\psi/v] &= \epsilon \\
w[\psi/v] &= \begin{cases} w & \text{if } w \neq v \\ \psi & \text{if } w = v \end{cases} \\
(\lambda w. \phi)[\psi/v] &= \begin{cases} \lambda w. (\phi[\psi/v]) & \text{if } w \neq v \\ \lambda w. \phi & \text{if } w = v \end{cases}
\end{aligned}$$

Furthermore, the function \mathcal{P} – used in the premisses of rules (*Loc*) and (*Let*) – is a function from nested environments NE (in Γ) to prefixes. It is defined in terms of the bullet operator (\bullet), which denotes concatenation of prefixes. The function \mathcal{P} is defined as follows:

$$\begin{aligned}
\mathcal{P}(\text{Loc}(\text{NAMES}, E) :: NE) &= \mathcal{P}(\Gamma) \bullet \text{NAMES} \# \\
\mathcal{P}(\text{Let}(\text{NAME}, E) :: NE) &= \mathcal{P}(\Gamma) \bullet \text{NAME} \\
\mathcal{P}(\epsilon) &= \epsilon
\end{aligned}$$

The *Basic* function from rule (*arrow/14*) takes a Standard ML shallow type and returns its most general deep type. This most general deep type ϕ will allow unification with any other deep type ψ , whose associated shallow type is identical to the shallow type of ϕ . Figure 5 shows two examples of the *Basic* function. Its definition is printed below. Note that in this definition ϕ and ϕ' denote fresh variables:

$$\begin{aligned}
\text{Basic}(\tau \rightarrow \tau') &= \phi \parallel \lambda v : \text{Basic}(\tau) \cdot \text{Basic}(\tau') \\
\text{Basic}(c) &= \phi \parallel \epsilon \\
\text{Basic}(v) &= \phi \parallel \phi'
\end{aligned}$$

Now that we have introduced all of the necessary notation, we can take a closer look at some of the most important rules of the operational semantics. Just as in [9] the set of rules has been split up in subsets following the syntactic categories of the language. We will discuss each one in turn, starting with the subset for atomic expression in Figure 6.

$$\begin{aligned}
& \Gamma(\text{longvid}) = (\sigma \& \psi, is) \\
& \quad \sigma \succ \tau \\
& \quad s = \mathcal{U}'(\phi, \psi) \\
& \frac{\mathcal{S} \text{ of } \Gamma = \mathcal{U}'\{\phi \mapsto s, \psi \mapsto s\}}{\Gamma \vdash \text{longvid} \Rightarrow \tau \& \phi} \text{ (longvid/2)} \\
& \frac{\Gamma \vdash \text{dec} \Rightarrow (\text{NAMES}, E, \sum_{k=1}^n \phi_k \parallel \epsilon) \\
& \quad \text{Let}(\mathcal{N} \text{ of } \Gamma, E) :: \Gamma \vdash \text{exp} \Rightarrow \tau \& \sum_{l=1}^m \psi_l \parallel \chi_l \\
& \quad \text{tynames } \tau \subseteq T \text{ of } \Gamma}{\Gamma \vdash \text{let dec in exp end} \Rightarrow \tau \& \sum_{k=1, l=1}^{n, m} \phi_k @ \psi_l \parallel \chi_l} \text{ (let/4)} \\
& \frac{\Gamma \vdash \text{exp} \Rightarrow \tau \& \phi}{\Gamma \vdash (\text{exp}) \Rightarrow \tau \& \phi} \text{ (bracket/5)}
\end{aligned}$$

Figure 6: Deep Type Semantics – Atomic Expressions

There are three rules for atomic expressions, the first of which deals with *longvid* expressions. It is important to note that in most cases, no unification will be necessary, because ϕ and ψ are already equal. When a situation like this occurs, the unification function \mathcal{U}' will yield the empty set. However, in the case of a recursive *longvid*, the unification function will yield a non-empty set which will be stored in the context Γ for future reference. At this point, note that even though the shallow types of recursive functions are finite, the deep types of recursive functions are not. We would therefore be unable to write them down without some form of recursion, but since we will need unification for function application anyway, constraint sets presented us with a logical candidate.

The second rule for atomic expressions is the rule for *let* expressions. The \mathcal{N} that is used to extend the nested environment NE , is extracted from the environment, where it has been stored by an instance of the (*equals/25*) rule, a little lower down in the derivation.

The third rule dealing with atomic expression is rather uninteresting, so we will move on to the expressions in Figure 7 instead. The first rule dealing with expression is one of the most complicated in the operational semantics. The reason it is so complicated is because the deep type of a higher-order function is dependent on the deep type of its argument(s). Consider for example the *map* function. Whether or not it will display any side effects cannot be determined by the deep type of *map* alone, but depends on the behaviour of its first argument. Figure 2 shows a (simplified) instance of the function application rule.

The second rule in the expression semantics set is not quite as complicated as the previous one, but requires some explanation nevertheless. It is obvious that a *handle* expression will display either the side effects of *exp* or those of *match*, but it is also important to register that the *handle* expression will never raise an *exn* exception as a result of *exp* raising one. If we would fail to do so, the assertion system would become unsound due to the derivability of something like: the expression *exp handles match* possibly throws an *exn* exception, because *exp* possibly throws one.

$$\frac{\begin{array}{l} \Gamma \vdash \text{exp} \Rightarrow \tau' \rightarrow \tau \ \& \sum_{k=1}^n \phi_k \\ \Gamma \vdash \text{atexp} \Rightarrow \tau' \ \& \sum_{j=1}^p \pi_j \\ \mathcal{S} \text{ of } \Gamma = \bigcup_{l=1, j=1}^{m, p} \{\varphi_l \mapsto s_{l,j}, \omega_l \mapsto s_{l,j}\} \end{array}}{\Gamma \vdash \text{exp} \ \text{atexp} \Rightarrow \tau \ \& \sum_{k=1}^n \phi'_k} \text{ (app/8)}$$

- where $\phi_k = \psi_k \parallel \sum_{l=1}^m \chi_l$
- and $\chi_l = \lambda x_l : \beta_l. \varphi_l \parallel \omega_l$
- and $\pi_j = \rho_j \parallel \alpha_j$
- and $s_{l,j} = \mathcal{U}'(\beta_l, \epsilon \parallel \alpha_j)$
- and $\phi'_k = \sum_{l=1, j=1}^{m, p} \rho_j \ @ \ \psi_k \ @ \ \varphi_l \parallel \omega_l$

$$\frac{\begin{array}{l} \Gamma \vdash \text{exp} \Rightarrow \tau \ \& \sum_{k=1}^n \phi_k \parallel \chi_k \\ \Gamma \vdash \text{match} \Rightarrow \text{exn} \rightarrow \tau \ \& \sum_{l=1}^m \psi_l \parallel \varphi_l \end{array}}{\Gamma \vdash \text{exp} \ \text{handle} \ \text{match} \Rightarrow \tau \ \& \sum_{k=1, l=1}^{n, m} (H(\text{exn}, \phi_k) \parallel \chi_k) + (\psi_l \parallel \varphi_l)} \text{ (handle/10)}$$

$$\frac{\Gamma \vdash \text{exp} \Rightarrow \text{exn}}{\Gamma \vdash \text{raise} \ \text{exp} \Rightarrow \tau \ \& T(\text{exn}) \parallel \epsilon} \text{ (raise/11)}$$

$$\frac{\Gamma \vdash \text{match} \Rightarrow \tau \ \& \phi}{\Gamma \vdash \text{fn} \ \text{match} \Rightarrow \tau \ \& \phi} \text{ (fn/12)}$$

Figure 7: Deep Type Semantics – Expressions

$$\frac{\begin{array}{l} \Gamma \vdash \text{mrule} \Rightarrow \tau \ \& \phi \\ \langle \Gamma \vdash \text{match} \Rightarrow \tau \ \& \psi \rangle \end{array}}{\Gamma \vdash \text{mrule} \langle | \ \text{match} \rangle \Rightarrow \tau \ \& \phi \langle + \psi \rangle} \text{ (match/13)}$$

Figure 8: Deep Type Semantics – Matches

The third and fourth rule dealing with expressions are rather uninteresting, so we will now continue with the set of rules dealing with matches. There is only one rule in this set, which can be found in Figure 8. The rule is fairly intuitive, but keep in mind that ψ can be a sum itself, which allows the semantics to deal with an arbitrary number of matches.

Another set with only one rule in it is the set of match rules in Figure 9. The match rule uses the aforementioned *Basic* function which takes a shallow type and returns the most general deep type matching the shallow one. This same match rule is also the sole creator of lambda expressions in deep types.

Now we come to the set of declarations in Figure 10. The two most important things happening in the first rule are

$$\frac{\begin{array}{l} \Gamma \vdash \text{pat} \Rightarrow (N, VE, \tau \ \& B) \\ \Gamma + VE \vdash \text{exp} \Rightarrow \tau' \ \& \phi \end{array}}{\Gamma \vdash \text{pat} \Rightarrow \text{exp} \Rightarrow \tau \rightarrow \tau' \ \& \epsilon \parallel \lambda N : B. \phi} \text{ (arrow/14)}$$

- where $B = \text{Basic}(\tau)$.

Figure 9: Deep Type Semantics – Match Rules

$$\frac{\begin{array}{l} \Gamma + U \vdash \text{valbind} \Rightarrow (NAME, VE, \sum_{k=1}^n \phi_k \parallel \chi_k) \\ VE' = \text{Clos}_{\Gamma, \text{valbind}} VE \\ U \cap \text{tyvars} VE' = \emptyset \end{array}}{\Gamma \vdash \text{val} \ \text{tyvarseq} \ \text{valbind} \Rightarrow (\{NAME\}, VE' \text{ in Env}, \sum_{k=1}^n \phi_k \parallel \epsilon)} \text{ (val/15)}$$

$$\frac{\begin{array}{l} \Gamma \triangleleft N_1 \vdash \text{dec}_1 \Rightarrow (NAMES, E_1, \sum_{k=1}^n \phi_k \parallel \epsilon) \\ \text{Loc}(N \text{ of } \Gamma, E_1) :: \Gamma \vdash \text{dec}_2 \Rightarrow (N_2, E_2, \sum_{l=1}^m \psi_l \parallel \epsilon) \end{array}}{\Gamma \vdash \text{local} \ \text{dec}_1 \ \text{in} \ \text{dec}_2 \ \text{end} \Rightarrow (\epsilon, E_2[N_2/N_1], \sum_{k=1, l=1}^{n, m} \phi_k \ @ \ \psi_l \parallel \epsilon)} \text{ (loc/21)}$$

- where N_1 is a fresh variable.

$$\frac{\begin{array}{l} \Gamma \vdash \text{dec}_1 \Rightarrow (NAMES_1, E_1, \phi \parallel \epsilon) \\ \Gamma \oplus E_1 \vdash \text{dec}_2 \Rightarrow (NAMES_2, E_2, \psi \parallel \epsilon) \end{array}}{\Gamma \vdash \text{dec}_1 \langle ; \rangle \text{dec}_2 \Rightarrow (NAMES_1 \cup NAMES_2, E_1 + E_2, \phi \ @ \ \psi \parallel \epsilon)} \text{ (decs/24)}$$

Figure 10: Deep Type Semantics – Declarations

turning *NAME* into a singleton and erasing the *future* term of *valbind*'s deep type. The grounds for the former are simply preparation for the union of name sets in rule (*decs/24*), while the latter is common sense: the future of the *valbind* expression is stored safely in the context Γ , but the declaration itself has lost all its potential when we put the keyword *val* in front of it.

The second rule dealing with declaration is the (*loc/21*) rule. Note that the nested environment *NE* gets extended with a *Loc* environment and that we have used a temporary variable N_1 during the evaluation of *decs*₁, which is replaced later on by the set of names from the body (*decs*₂) of the *local* declaration.

The final rule in the declarations set is interesting, but not very difficult to read. Remember that declarations have no future, hence the ϵ term in both premisses and the conclusion.

The set of value bindings in Figure 11 contains two rules. Both of them differ from the way they were defined in [9] in that they return a three tuple instead of just the *VE*. The reason for the presence of the first element has been demonstrated in the rule for declarations (*loc/21*) and atomic expressions (*let/4*). The reason for the third element, which holds the deep type, is less obvious especially because we are not even keeping track of the shallow type in this rule. The following example demonstrates the importance of remembering the deep type in value bindings and consequently, declarations:

```
val r = ref 5;
val x = !r;
val y = r := !r + 1;
val z = !r
```

We somehow want to capture the fact that this little program has quite a few side effects and that its deep type is

$$\frac{\Gamma \vdash pat \Rightarrow (N, VE, \tau \& \phi) \quad \Gamma \triangleleft N \vdash exp \Rightarrow \tau \& \phi}{\Gamma \vdash pat = exp \Rightarrow (N, VE, \phi)} \text{ (equals/25)}$$

$$\frac{\Gamma + VE \vdash valbind \Rightarrow (NAME, VE, \phi)}{\Gamma \vdash \mathbf{rec} \text{ valbind} \Rightarrow (NAME, VE, \phi)} \text{ (rec/26)}$$

Figure 11: Deep Type Semantics – Value Bindings

$$\frac{vid \notin Dom(\Gamma) \text{ or } is \text{ of } \Gamma(vid) = v}{\Gamma \vdash vid \Rightarrow (vid, \{vid \mapsto (\tau \& \phi, v)\}, \tau \& \phi)} \text{ (vid/34)}$$

$$\frac{\Gamma \vdash pat}{\Gamma \vdash (pat)} \text{ (brack/37)}$$

Figure 12: Deep Type Semantics – Atomic Patterns

a composition of the four deep types that represent \mathbf{r} , \mathbf{x} , \mathbf{y} and \mathbf{z} . To be able to this, we need the deep type information to be present in both the *valbind* rules and the ones for declarations.

The last set which has been adapted from [9] is that of atomic patterns in Figure 12. There are two rules in this set, only one of which is of particular significance. This rule, which is called (*vid/34*), is the axiom of deriving a *VE*. It also remembers the variable name and its shallow and deep type.

The set of references in Figure 13 is the first set which does not have a counterpart in [9]. As mentioned before, [9] only deals with general language constructs and not with any specific functions. However, since only specific functions have the ability of exhibiting side effects, our systems will need rules dealing with these.

Figure 13 shows the set of rules dealing with references. Both rules use a *PREFIX* in front of the *vid*. This prefix determines the absolute location in the (nested) environment of the *vid* as described in Section 2.1.1 and Figure 3.

The set of rules for inferring references in Figure 14 is the last of the operational semantics subsets. It basically describes how to infer the premisses of the rules dealing with references in Figure 13 and uses the aforementioned function \mathcal{P} to derive the remaining part of the prefix once it has located the reference *vid*.

2.2 The Deep Type Constraints System

$$\frac{\Gamma \vdash vid \Rightarrow (\tau \mathbf{ref}, \mathbf{PREFIX})}{\Gamma \vdash vid := exp \Rightarrow \mathbf{unit} \& U(\mathbf{PREFIX} \text{ vid}) || \epsilon} \text{ (update)}$$

$$\frac{\Gamma \vdash vid \Rightarrow (\tau \mathbf{ref}, \mathbf{PREFIX})}{\Gamma \vdash !vid \Rightarrow \tau \& R(\mathbf{PREFIX} \text{ vid}) || \epsilon} \text{ (deref)}$$

Figure 13: Deep Type Semantics : References

$$\frac{E \vdash vid \Rightarrow \tau \mathbf{ref}}{Loc(\mathbf{NAMES}, E) :: \Gamma \vdash (\tau \mathbf{ref}, \mathcal{P}(RE \text{ of } \Gamma) \bullet \mathbf{NAMES}\#)} \text{ (Loc)}$$

$$\frac{\Gamma \vdash !vid \Rightarrow (\tau \mathbf{ref}, \mathbf{PREFIX}) \quad vid \notin Dom(E)}{Loc(\mathbf{NAMES}, E) :: \Gamma \vdash vid \Rightarrow (\tau \mathbf{ref}, \mathbf{PREFIX})} \text{ (Loc')}$$

$$\frac{E \vdash vid \Rightarrow \tau \mathbf{ref}}{Let(\mathbf{NAME}, E) :: \Gamma \vdash (\tau \mathbf{ref}, \mathcal{P}(RE \text{ of } \Gamma) \bullet \mathbf{NAME}.)} \text{ (Let)}$$

$$\frac{\Gamma \vdash !vid \Rightarrow (\tau \mathbf{ref}, \mathbf{PREFIX}) \quad vid \notin Dom(E)}{Let(\mathbf{NAME}, E) :: \Gamma \vdash vid \Rightarrow (\tau \mathbf{ref}, \mathbf{PREFIX})} \text{ (Let')}$$

$$\frac{RE \text{ of } \Gamma = \epsilon \quad C \text{ of } \Gamma \vdash vid \Rightarrow (\tau \mathbf{ref}, \mathbf{PREFIX})}{\Gamma \vdash vid \Rightarrow (\tau \mathbf{ref}, \mathbf{PREFIX})} \text{ (Empty)}$$

Figure 14: Deep Type Semantics – Inferring References

We will now describe the other half of the deep type system: the constraint system for deep types. Its syntax can be found in Figure 15 and its semantics in Figures 16, 17 and 18.

2.2.1 Basic Assertions

The first question which we will have to answer is where in the program we want to put the assertions. The answer will once again come from the analogy with Standard ML's shallow type system: since the assertions describe a constraint system, and the shallow constraints are specified in a structure's signature, the signature would be a logical place for our assertions as well.

The basic assertions of the deep type constraint system deal with side-effects such as updating and accessing variables (*m updates ref sf* and *m reads ref sf*) and throwing exceptions (*m throws exn sf*). One can combine these side-effect statements by means of the boolean operators *and* (\wedge), *or* (\vee) and *not* (**not**).

The letter *m* which you can see at the start of every assertion is a modal operator indicating the specified property is one that will necessarily be displayed (\square – pronounced as *box*) or merely a possibility (\diamond – pronounced as *diamond*).

The subject of these and all other assertions are the *simple functions* or *sf* expressions. A simple function is a function identifier possibly together with one or more arguments or argument placeholders. An argument placeholder can be either an underscore, which denote a *don't care* argument or a ghost variable, which may impose some constraints on the argument. Ghost variables will be discussed further in the next section. In the deep type assertion system, a function identifier does not necessarily need to be supplied with all of its arguments, but since assertions only range over the *past*

```

spec ::= asrt ⟨spec⟩
asrt ::= assert basic ⟨where basic⟩
      | local lcls in spec end
lcl ::= var gv where basic
lcls ::= lcl ⟨lcls⟩
basic ::= isPure sf
      | m atom sf
      | not basic
      | basic ∧ basic
      | basic ∨ basic
atom ::= throws exn
      | updates ref
      | reads ref
      | isPure
m ::= <>
   | []
sf ::= fid ⟨sf⟩
fid ::= gv
      | v
      | -
gv ::= .v
ref ::= v.ref
      | v#ref
      | v
v ::= longvid

```

Figure 15: Syntax of Deep Type Assertions

term of a deep type, more arguments usually means more information.

However, if you would like to make sure a function with two abstractions always shows a certain side effect immediately after its first one has been instantiated, an assertion over the partially applied function would be the sensible option. An example of this case is the function `f` together with both the assertion which has been supplied with only one argument, and the one which has been supplied with two:

```

SML> val f = fn x => (!c ; fn y => !d)

DTS> assert [] reads c (f _)
DTS> assert [] reads c (f _ _)

```

Although both assertions will evaluate to the same truth value (i.e. `true`), the former gives us more specific information. We will now continue with the description of some advanced assertions.

2.2.2 Ghost Functions and Variables

The concept of ghost variables is described in detail in [2].

As mentioned in the previous paragraph, ghost variables may be used in simple functions. However, they have to be declared first for this to be possible. There are two ways to define a ghost variable. One is by specifying a `local` environment, the other is in a `where` expression. Either way allows you to specify a *basic* expression on the variable in question. Once a ghost variable has been declared, it may be used in a simple function expression, to which it will

feed information about its deep type. Consider the following example:

```

SML> val f = fn h => fn i => h i

DTS> local var _g where <> throws e (_g _)
DTS> in   assert <> throws e (f _g _)
DTS> end

```

In this fragment, we have declared a ghost function `_g` and used it as the argument of the identity function `f` in the assertion in the body of the `local` statement. The validity of this assertion will be derived, by the deep type inference system, from the information supplied about the ghost variable `_g` and the deep type of the identity function `f`.

Ghost variables boost the expressiveness of deep type assertion considerably and therefore play a major role in specifying constraints on deep types. They could even be helpful during program design, as we will see in Section 3.3.

One could also imagine ghost variables ranging over properties instead of simple functions. This would significantly increase reusability of deep type code, especially in the case of checking for program consistency (Section 3.2). Although the feature of ghost variables ranging over *properties* has not been included in this version of the deep type system for Standard ML, it should not be extremely difficult to add.

2.2.3 The Operational Semantics

We have now arrived at the point at which we can introduce the semantics of assertions shown in Figures 16 to 18. These three figures encapsulate what it means for an assertion to be true, or alternatively, whether a compiler would accept or reject the program which is the subject of the assertion. Figure 16 shows the basic assertion semantics, up to the *models* relation. The rules are fairly straightforward; an assertion A describing a property p is true if and only if its subject sf , has deep type ϕ and ϕ is a witness of the property p . It is necessary to pass both the deep type ϕ and the constraint set \mathcal{S} from the context Γ to the *models* relation, because ϕ alone does not provide sufficient information. That is, it may contain variables on which restrictions have been imposed, and subsequently stored in \mathcal{S} .

Furthermore, the VE function holds the ghost variables and their properties and in order to accommodate the `not` operator, the rules keep track of a boolean b , whose value will determine the polarity of the \models -relation to be derived in the $(atom_n)$ and $(pure_n)$ rules. Also note that \uplus in the (and) -rule in Figure 16 denotes range union. That is, if $VE = \{x \mapsto A\}$ and $VE' = \{x \mapsto B\}$ then $VE \uplus VE' = \{x \mapsto A \cup B\}$.

The *models* relation in Figure 17 is not a very complicated one. The most striking feature of this relationship is probably the difference in evaluation between *atom* and *isPure* expressions. The cause of this divide is a polarity discrepancy: *atom* expressions make positive statements of the form ‘function f throws exception e ’, whereas an *isPure* expression makes a negative statement such as ‘function g does not have any side effects’. Of course we can also compose

$$\begin{array}{c}
\frac{}{(\epsilon \parallel \epsilon, \mathcal{S}) \models \text{isPure}} \text{(pure)} \\
\\
\frac{\Gamma \vdash \text{asrt } \langle \Gamma \vdash \text{spec} \rangle}{\Gamma \vdash \text{asrt } \langle \text{spec} \rangle} \text{(spec)} \\
\\
\frac{\Gamma \langle +VE \rangle \vdash \text{be} \Rightarrow (\text{true}, VE') \quad \langle \Gamma \vdash \text{be} \Rightarrow (\text{true}, VE) \rangle}{\Gamma + VE' \vdash \text{assert } \text{basic} \langle \text{where } \text{basic} \rangle} \text{(assert)} \\
\\
\frac{\Gamma \vdash \text{lcls} \Rightarrow (\text{true}, VE) \quad \Gamma + VE \vdash \text{spec}}{\Gamma \vdash \text{local } \text{lcls} \text{ in } \text{spec} \text{ end}} \text{(local)} \\
\\
\frac{\Gamma \vdash \text{basic} \Rightarrow (b, VE) \quad \langle \Gamma + VE \vdash \text{lcls} \Rightarrow (b, VE') \rangle}{\Gamma \vdash \text{var } gv \text{ where } \text{basic} \langle \text{lcls} \rangle \Rightarrow (b, VE')} \text{(var)} \\
\\
\frac{\Gamma \vdash sf \Rightarrow \tau \& \phi \quad (\phi, \mathcal{S} \text{ of } \Gamma) \models m \text{ atom}}{\Gamma \vdash m \text{ atom } sf \Rightarrow (\text{true}, \{sf \mapsto m \{atom\}\})} \text{(atom}_1\text{)} \\
\\
\frac{\Gamma \vdash sf \Rightarrow \tau \& \phi \quad (\phi, \mathcal{S} \text{ of } \Gamma) \not\models m \text{ atom}}{\Gamma \vdash m \text{ atom } sf \Rightarrow (\text{false}, \{sf \mapsto m \{atom\}\})} \text{(atom}_2\text{)} \\
\\
\frac{\Gamma \vdash sf \Rightarrow \tau \& \phi \quad (\phi, \mathcal{S} \text{ of } \Gamma) \models \text{isPure}}{\Gamma \vdash \text{isPure } sf \Rightarrow (\text{true}, \{sf \mapsto \text{isPure}\})} \text{(pure}_1\text{)} \\
\\
\frac{\Gamma \vdash sf \Rightarrow \tau \& \phi \quad (\phi, \mathcal{S} \text{ of } \Gamma) \not\models \text{isPure}}{\Gamma \vdash \text{isPure } sf \Rightarrow (\text{false}, \{sf \mapsto \text{isPure}\})} \text{(pure}_2\text{)} \\
\\
\frac{\Gamma \vdash \text{basic} \Rightarrow (\neg b, VE)}{\Gamma \vdash \text{not } \text{basic} \Rightarrow (b, VE)} \text{(not)} \\
\\
\frac{\Gamma \vdash \text{basic}_1 \Rightarrow (b_1, VE) \quad \Gamma \vdash \text{basic}_2 \Rightarrow (b_2, VE')}{\Gamma \vdash \text{basic}_1 \wedge \text{basic}_2 \Rightarrow (b_1 \wedge b_2, VE \uplus VE')} \text{(and)} \\
\\
\frac{\Gamma \vdash \text{basic}_1 \Rightarrow (b_1, VE) \quad \Gamma \vdash \text{basic}_2 \Rightarrow (b_2, VE)}{\Gamma \vdash \text{basic}_1 \vee \text{basic}_2 \Rightarrow (b_1 \vee b_2, VE)} \text{(or)}
\end{array}$$

Figure 16: Assertion Semantics

$$\begin{array}{c}
\frac{}{(\epsilon \parallel \epsilon, \mathcal{S}) \models \text{isPure}} \text{(pure)} \\
\\
\frac{(\phi_1 \parallel \phi_3, \mathcal{S}) \models \text{isPure} \quad (\phi_2 \parallel \phi_3, \mathcal{S}) \models \text{isPure}}{(\phi_1 + \phi_2 \parallel \phi_3, \mathcal{S}) \models \text{isPure}} \text{(pplus}_{\text{pure}}\text{)} \\
\\
\frac{(\phi_1 \parallel \phi_3, \mathcal{S}) \models \square A \quad (\phi_2 \parallel \phi_3, \mathcal{S}) \models \square A}{(\phi_1 + \phi_2 \parallel \phi_3, \mathcal{S}) \models \square A} \text{(pplus}\square\text{)} \\
\\
\frac{(\phi_1 \parallel \phi_3, \mathcal{S}) \models \diamond A}{(\phi_1 + \phi_2 \parallel \phi_3, \mathcal{S}) \models \diamond A} \text{(pplus}\diamond_l\text{)} \\
\\
\frac{(\phi_2 \parallel \phi_3, \mathcal{S}) \models \diamond A}{(\phi_1 + \phi_2 \parallel \phi_3, \mathcal{S}) \models \diamond A} \text{(pplus}\diamond_r\text{)} \\
\\
\frac{(\phi_1 \parallel \phi_3, \mathcal{S}) \models A \quad A \neq \text{isPure}}{(\phi_1 @ \phi_2 \parallel \phi_3, \mathcal{S}) \models A} \text{(@}_l\text{)} \\
\\
\frac{(\phi_2 \parallel \phi_3, \mathcal{S}) \models A \quad A \neq \text{isPure}}{(\phi_1 @ \phi_2, \mathcal{S}) \parallel \phi_3 \models A} \text{(@}_r\text{)} \\
\\
\frac{(\phi_1 \parallel \phi_3, \mathcal{S}) \models \text{isPure} \quad (\phi_2 \parallel \phi_3, \mathcal{S}) \models \text{isPure}}{(\phi_1 @ \phi_2, \mathcal{S}) \parallel \phi_3 \models \text{isPure}} \text{(@}_{\text{pure}}\text{)} \\
\\
\frac{v \notin \text{Dom}(\mathcal{S})}{(v, \mathcal{S}) \models \diamond \text{atom}} \text{(\diamond}_{\text{var}_{\text{atom}}}\text{)} \\
\\
\frac{}{(T(\text{exn}) \parallel \phi, \mathcal{S}) \models \text{throws } \text{exn}} \text{(T)} \\
\\
\frac{\text{ref} \Rightarrow r}{(U(\text{ref}) \parallel \phi, \mathcal{S}) \models \text{updates } r} \text{(U)} \\
\\
\frac{\text{ref} \Rightarrow r}{(R(\text{ref}) \parallel \phi, \mathcal{S}) \models \text{reads } r} \text{(R)} \\
\\
\frac{(\phi, \mathcal{S}) \models A}{(\phi, \mathcal{S}) \models \square A} \text{(\square I)} \\
\\
\frac{(\phi, \mathcal{S}) \models A}{(\phi, \mathcal{S}) \models \diamond A} \text{(\diamond I)} \\
\\
\frac{(\phi[v'/v], \mathcal{S}) \models A}{(v, \mathcal{S}) \models A} \text{(var)}
\end{array}$$

- where $\phi = \mathcal{S}(v)$
- and $v' \notin \text{Dom}(\mathcal{S})$ and $v' \notin \phi$

Figure 17: Deep type *models* relation

$$\frac{}{\text{NAME} \Rightarrow \text{NAME}} \quad (1)$$

$$\frac{\text{ref} \Rightarrow r}{\text{NAME.ref} \Rightarrow \text{NAME.r}} \quad (2)$$

$$\frac{v \in \text{NAMES} \quad \text{ref} \Rightarrow r}{\text{NAMES}\#\text{ref} \Rightarrow v.r} \quad (3)$$

Figure 18: $\text{ref} \Rightarrow r$ relation

negative statements without the `isPure` keyword, because we have `not`. The difference is that the keyword `not` will never reach the `models` relation, because it is taken care of by the assertion semantics in Figure 16. The `isPure` keyword however, will reach the `models` relation and therefore requires different rules for the `+` and `@` operators.

Another important item to mention is the presence of the constraint set \mathcal{S} . We mentioned before that \mathcal{S} is needed in the `models` relation for its information on possible variables in the deep type ϕ . We are now at the point where we can see how it is used. Most rules from the `models` relation in Figure 17 just pass \mathcal{S} up to their premisses. Only the rules ($\diamond \text{var}_{atom}$) and (var) actually use the constraint set. The former captures the fact that any side effect is possible when the deep type consists of a variable on which no constraints have been imposed. The latter deals with variables for which there *is* a constraint set present in \mathcal{S} by inserting these constraints into the deep type and proving the property A of this new deep type.

Also, note that there can never be more than one modal operator in front of a `atom` assertion, which ensures the validity of the rules ($\square I$) and ($\diamond I$). Furthermore, the rule for variables (`var`) could potentially be applied an infinite number of times in the case of a recursive function, were it not for the crucial substitution $[v'/v]$ of any recursive variables. This prevents the (`var`) rule from being applied to the same variable more than once.

Finally, the reference reduction in Figure 18 is relatively intuitive. It simply reduces a deep type reference, which may have sets of identifiers in its prefix, to an assertion reference, which does not.

2.2.4 Possible Extensions

The current set of assertions has a limited expressiveness, but deep types allow other, more complicated assertions, to be checked as well. Because our deep types preserve the order in which the side effects occur, one could think of an assertion stating that if program raises a certain exception, some reference r will not be updated. Another example would be to require a program to always read a certain variable before updating it.

Furthermore, in Standard ML exceptions and functions on references are responsible for only a fraction of its side effect. If we were to extend the deep types to include other side effects like reading from and writing to files, opening and closing channels or accessing database and system in-

```

structure PV :
  sig
    val succ : int -> int
    assert not (<> throws Overflow (succ _))

    val succ' : (int -> int) -> int -> int
    assert <> throws Overflow (succ' _ _)

    val inc : unit -> int
    assert [] updates inc#c (inc _)

    val get : unit -> int
    assert not (<> updates get#c (get _))
  end
=
struct
  fun succ x =
    x + 1 handle Overflow => x

  fun succ' round x =
    (round x) + 1 handle Overflow => (round x)

  local
    val c = ref 0
  in
    fun inc () = (c := !c + 1 ; !c)
    fun get () = !c
  end
end

```

Figure 19: Program Verification – Example Program

formation, the number of practical application scenarios of the assertion system would soar.

3. PRACTICE

Now that we have formally introduced the deep type system, we will go back to our initial motivation for introducing it. We set out to create a formal system which would narrow the type information gap that arises when imperative features are added to a functional languages, and which would aid the programmer to accomplish increased program safety. The resulting system has indeed narrowed the gap. The fact that it does indeed aid the programmer to accomplish increased program safety in several ways, will be shown in the sequel.

3.1 Program Verification

The most obvious application of the deep type system is of course program verification. By means of defining assertions of various levels of complexity, one can force a function to have any number of properties. Take for example the structure `PV` in Figure 19. It consists of four different functions `succ`, `succ'`, `inc` and `get`. We have added exactly one assertion for each of these functions to the signature of `PV` to demonstrate the basic functionality of the assertion system, but there is of course no restriction on the number of assertions per function.

The first assertion states that it is impossible for the function `succ` to throw an `Overflow` exception, which is obvi-

ously true. The second assertion claims that unlike `succ`, the function `succ'` may sometimes throw an `Overflow` exception, because we do not know anything about its first argument `round`. `Round` may exhibit all sorts of side effects, including throwing an `Overflow` exception. The third assertion, states that the function `inc` always updates its local variable `c`, whereas the fourth one declares that the function `get` never does. It should be obvious that both of these assertions will evaluate to `true`.

3.2 Program Consistency

Another useful application of the deep system is as a program consistency checker. Software is being updated all the time, and the more code it involves, the more difficult it becomes to check that the updated part of the software (or function) still has the same behaviour as the original one. Of course the deep type system will not be able to make any statements about the behaviour of a function with regard to specific values, but unlike Standard ML's regular type system, it will be able to provide information about the function's side-effects. The deep type system can ensure that the new version of the function still exhibits all the desired side-effects by subjecting it to the same constraints as the original function.

As an example, consider the two functions which are shown in Figure 20. We cannot make sure that functions `i2b_old` and `i2b_new` denote the same function, but we can make sure that the new version never throws an `Overflow` exception and that it always updates the boolean reference `b`. Even though for an example of this complexity one could just check these properties by hand, a Standard ML program does not need to increase very much in size to become sufficiently complicated for manual deep type checking to become virtually impossible. Therefore, the deep type system should be particularly useful for the development and maintenance of large programs. For this reason, we believe that deep types represent a scalable program-checking technology which is comparable with approaches such as extended static checking [5].

One thing which we would like to point out about program consistency applications of the deep type system is that they would benefit enormously from the introduction of ghost variables ranging over properties instead of deep types as mentioned in Section 2.2.2. If the programmer would be able to assign names to properties of arbitrary complexity, the reusability of deep type code would go up and therefore make the deep type system scale better for larger applications.

3.3 Aid in Software Development

The third and most advanced application of the deep type system makes use of the aforementioned ghost variables. In the deep type system it is possible to apply a function to an argument which is not actually implemented (yet). An argument like this can be specified by means of a ghost variables and the programmer can enforce it to have any desired properties.

Take for example the well known function `map` for example. Its first argument is a function, its second argument a list and it results in a list of which the function has been applied to every element. We cannot prove that `map` is a pure

```
structure PC :
  sig
    val i2b_old : int -> bool
    assert not (<> throws Overflow (i2b_old _))
    assert [] updates i2b_old#b (i2b_old _)

    val i2b_new : int -> bool
    assert not (<> throws Overflow (i2b_new _))
    assert [] updates i2b_new#b (i2b_new _)
  end
=
struct
  local
    val b = ref false;
    fun f 0 = false
      | f 1 = true
      | f _ = raise Overflow
    fun g 0 = (b := false ; false)
      | g 1 = (b := true ; true)
      | g _ = (b := false ; raise Overflow)
  in
    fun i2b_old i =
      (b := f i ; !b)
      handle Overflow => (b := false ; false)
    fun i2b_new i =
      g i handle Overflow => (b := false ; false)
  end
end
```

Figure 20: Program Consistency – Example Program

```
structure S :
  sig
    val c : string ref

    val higher : (string -> unit) -> string -> unit
    assert <> updates c (higher _ _)
    assert [] updates c (higher _f _)
    where [] updates c (_f _)
  end
=
struct
  val c = ref ""
  local
    fun helper s = c := !c ^ s
  in
    fun higher f s =
      case (explode s) of
        #("::tail => helper tail
        | #{"::tail => helper tail
        | #"<::tail => helper tail
        | _ => f s
  end
end
```

Figure 21: Aid in Software Development – Example Program

function, because its first argument, a function, may have any number of side effects about which we know nothing until the actual function is supplied to `map`. However, we do know that if the argument function is a pure function, then `map` will be pure as well. The deep type system allows us to specify this very statement in the assertion system in the following way:

```
assert isPure (map _f _)
  where isPure (_f _)
```

The programmer now knows that if he should decide to implement a pure function to feed the `map` function, `map` itself would be pure as well. Using ghost variables in this way can save the programmer quite a lot of time, because the only other alternative would be to implement a specific function, construct an assertion to make sure it is pure, feed it to `map` and construct another assertion to ensure that `map` together with the specified function is pure as well. All this work will not even give you the same amount of information as the assertion with the ghost variable, because the only information it provides you with, is that `map` is pure when fed with that one specific function, which just happens to be pure.

Figure 21 shows another example of an assertion with a ghost variable. The idea is the same as with the `map` function: if we give the `higher` function two unspecified arguments, the best information we can get is that of the possibility of an update of `c`, but if we use a ghost variable, we are allowed to command the `higher` function to always update the reference `c`.

These two examples have shown that assertions with ghost variables allow us to manufacture a deep type driven method of developing programs, instead of writing an implementation and then later on specifying what properties we would like it to have and check for those. Once again we see a parallel between shallow types and deep types, because shallow signatures are quite often specified before an implementation is written, as a way of giving an abstract formalisation of the problem.

3.4 Future Applications

The examples we have seen in this section are all taken from the Standard ML fragment as described in Figure 4, which is very limited in terms of practical applications because it lacks certain key language features like datatype declaration, non-atomic patterns and exception declarations. It also fails to include any library functions, which rules out the majority of potential practical applications.

However, if we were to extend the fragment to cover the whole of Standard ML, possibly even including the libraries or concurrent extensions, there is no limit to the number of applications of the deep type system. We already discussed what the extension described in Section 2.2.4 could do to the expressiveness of the assertion system, but we have not even mentioned any applications in concurrent ML yet. Concurrent systems are a well known niche for type and effect systems and because of the close connection between the deep type system and type and effect systems, concurrent systems could potentially benefit significantly from an

implementation of our deep type system.

4. RELATED WORK

The present work hopes to draw inspiration from significant interesting work on the boundary between the traditional static type inference which is well-known from functional programming languages and the work on extended static checking as typically applied to imperative or object-oriented languages.

Extended static checking in languages such as ESC/Java [5] uses assertions in program code to supplement strong static type checking. The purpose of these, as in the present work, is to find programmer errors which would normally go undetected until the program was executed. In type-safe languages faulty programs can fail with run-time exceptions such as (in the case of Java) a null pointer or class cast exception; or (in Standard ML) an integer overflow exception; or (in either) an array out-of-bounds exception. None of these failures are really acceptable and the ideal way to detect them is statically, when the developer is best able to efficiently detect and repair their programming errors.

Protocol types as used in Vault [4] and similar approaches such as *scoped methods* [14] also help programmers to eliminate unacceptable combinations of method invocations, preventing errors of procedure such as attempting to write to a closed file. We would like to see some of the flavour of this work come into work on deep types in the future.

Among the better-known approaches to detecting programmer errors in functional programming languages include the use of dependent types, even as applied to impure functional languages such as Standard ML [16]. However, an improved understanding of the relationships between functions with local state and objects, as found through [12], can help us to apply variants of the OO extended static checking methods directly to impure functional languages such as Standard ML, and this is a possible direction for future work on the present project.

Our work on deep types has some similarities with the Extended ML formal program development method [13] but the differences are more pronounced. Extended ML includes axioms in signatures for the purpose of refining an initial abstract specification into an efficiently executable implementation, with each step in the development supported by a formal proof of correctness. The intention is that the full formal proof of correctness guarantees freedom from execution errors of all kinds under normal operational execution. In contrast the assertions which we include in signatures lead to no such guarantees of program correctness but they do not incur the same penalties in development costs as full program verification.

5. CONCLUSIONS

We have introduced a deep type system for Standard ML. This is an extended type system which not only assigns a functional type to every function by means of shallow typing, but also captures the side effect behaviour of a program by means of the deep typing. A deep type system allows us to differentiate side-effecting procedures from pure functions, a

difference which is masked by Standard ML's shallow type system.

The deep type system was defined in analogy with Standard ML's shallow type system and therefore consists of both a type inference system, and a way of imposing constraints on the deep types. We have given a formal description of the both of these systems and explained most of the ideas behind them.

We then presented some practical examples which not only demonstrated how to use the deep type system, but also showed its potential when used in programs of a reasonable size. Some extensions were suggested to make the deep type system more expressive and more suitable for practical applications.

In the future, we hope to implement the deep type system for Standard ML by integrating it into one of the compilers, provide deep type support for the libraries with side effects and perhaps extend the deep type system to cover Concurrent ML.

6. ACKNOWLEDGMENTS

The authors would like to thank John Longley and Don Sannella for their comments on this paper.

Stephen Gilmore is supported by the Mobile Resource Guarantees project (MRG, project IST-2001-33149). The MRG project is funded under the Global Computing pro-active initiative of the Future and Emerging Technologies part of the Information Society Technologies programme of the European Commission's Fifth Framework Programme.

Laura Korte is supported by a studentship from the Laboratory for Foundations of Computer Science of The University of Edinburgh.

7. REFERENCES

- [1] T. Amtoft, F. Nielson, and H. R. Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, 1999.
- [2] M. Clint and C. Vicent. The use of ghost variables and virtual programming in the documentation and verification of programs. *Software—Practice and Experience*, 14(8):711–737, 1984.
- [3] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM Press, 1982.
- [4] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 59–69, June 2001.
- [5] C. Flanagan, K. Rustan M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37, pages 234–245. ACM SIGPLAN Notices, May 2002.
- [6] S. Gilmore. Designing for proof. In *Mathematics of Software Quality*, pages 1–15, 1995.
- [7] S. Gilmore. Deep type inference for mobile functions. In P. Trinder and G. Michaelson, editors, *Proceedings of the First Scottish Functional Programming Workshop*. Inspect, Aug. 1999.
- [8] X. Leroy and F. Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, 22(2):340–377, 2000.
- [9] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (revised)*. MIT Press, 1997.
- [10] F. Nielson and H. R. Nielson. Type and effect systems. In *Correct System Design*, pages 114–136, 1999.
- [11] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis: Flows and Effects*. Springer, 1999.
- [12] A. Pitts and I. Stark. Operational reasoning for functions with local state. In Gordon and Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 227–273. Publications of the Newton Institute, Cambridge University Press, 1998.
- [13] D. Sannella. Formal program development in extended ml for the working programmer. In *Proc. 3rd BCS/FACS Workshop on Refinement*. Springer, 1990.
- [14] G. Tan, X. Ou, and D. Walker. Enforcing resource usage protocols via scoped methods. In G. Ghelli, editor, *Proceedings of the Tenth International Workshops on Foundations of Object-Oriented Languages (FOOL 10)*, New Orleans, Jan. 2003.
- [15] A. K. Wright. Simple imperative polymorphism. *Lisp Symb. Comput.*, 8(4):343–355, 1995.
- [16] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In K. D. Cooper, editor, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pages 249–257, Montreal, Canada, June 1998. ACM Press.