The 2006 Federated Logic Conference

The Seattle Sheraton Hotel and Towers

Seattle, Washington

August 10 - 22, 2006

IJCAR '06 Workshop

PLPV '06:
Programming Languages meets Program Verification

August 21st, 2006

Proceedings

Editors:

A. Stump, H. Xi

Preface

This volume contains informal proceedings for the Programming Languages meets Program Verification (PLPV '06) workshop, held August 21, 2006. PLPV is an affiliated workshop of the Third International Joint Conference on Automated Reasoning (IJCAR '06), part of the 2006 Federated Logic Conference (FLoC). The main goal of PLPV is to bring together researchers exploring language-based approaches to program verification. Typically in these approaches, the programming language provides mechanisms which allow the programmer to express both specificational and verificational annotations. One commonly used mechanism is dependent types, where specifications are expressed as types, and the programming language allows proofs of specifications to be expressed as terms inhabiting those types.

These proceedings contain seven regular research papers and two short position papers. These papers touch on topics in Martin-Löf type theory, verified programming in Haskell, and environments and languages for verified programming. In addition to these papers, we received three other submissions, of which one was withdrawn and the other two were not accepted for inclusion in the workshop. Papers were received and reviewed using the excellent EasyChair system.

We were delighted Peter Dybjer agreed to be the workshop's (sole) invited speaker, contributing a talk with the same title as the workshop's. A panel discussion on the future of research in the field is also part of the program.

For their support of our organization of this workshop, we would like to thank the FLoC workshop chair Gopal Gupta and the IJCAR workshop chair Maria Paola Bonacina. We would like to thank Tom Ball and others in the FLoC organizing team for their organizational work, and Andrei Voronkov and those working with him to provide EasyChair. We would very much like to thank the members of our excellent program committee, as well as their external reviewers, for detailed and thorough reviewing of the submitted papers: Thorsten Altenkirch, Hugo Herbelin, Simon Peyton-Jones, Randy Pollack, Carsten Schuermann, Zhong Shao, and Tim Sheard. Finally, we are grateful to all authors of submitted papers, and to all those who will have attended the workshop. We are confident your efforts will continue to advance the state of the art of this growing field.

Aaron Stump and Hongwei Xi
PLPV co-chairs
St. Louis and Boston, July 2006

Contents

Programming Languages Meet Program Verification (Invited Talk)

Peter Dybjer
Chalmers University of Technology

Abstract

I will begin by giving an overview of the CoVer project (Combining Verification Methods in Software Development, 2003-2005) at Chalmers University. This was a research project comprising both researchers in Programming Languages (especially in functional programming) and Program Verification (especially in random testing, automatic theorem proving, type theory, and proof assistants). The goal of this project was to provide an environment for Haskell programming which provides access to tools for automatic and interactive correctness proofs as well as to tools for testing. I will both say something about the contributions and about the difficulties we had in fully realizing our goal. I will also give my view of some of the main scientific issues underlying the project. In particular I will say something about why it became important to understand the relationship between the "integrated" (propositions-as-types-based) and "external" approach to program verification. In the second part of the talk I will illustrate this issue by an example: the proof of correctness of a Haskell program which normalizes terms and types in Martin-Lf type theory with one universe.

Constructive membership predicates as index types

James Caldwell *and Josef Pohl †
Department of Computer Science
University of Wyoming
Laramie, WY

Abstract

In the constructive setting, membership predicates over recursive types are inhabited by terms indexing the elements that satisfy the criteria for membership. In this paper, we motivate and explore this idea in the concrete setting of lists and trees. We show that the inhabitants of membership predicates are precisely the inhabitants of a generic shape type. We show that membership of x (of type T) in structure S, $(x \in_T S)$ can not, in general, index all parts of a structure S and we generalize to a form $\rho \in S$ where ρ is a predicate over S. Under this scheme, $(\lambda x.True) \in S$ is the set of all indexes into S, but we show that not all subsets of indexes are expressible by strictly local predicates. Accordingly, we extend our membership predicates to predicates that retain state "from above" as well as allow "looking below". Predicates of this form are complete in the sense that they can express every subset of indexes in S. These ideas are motivated by experience programming in Nuprl's constructive type theory and the theorems for lists and trees have been formalized and mechanically checked.

1 Introduction

Recently, there has been significant interest in combining techniques of formal constructive proof with programming type systems and language based approaches to verification [28]; witness this workshop. Some of the most notable recent efforts include the Omega programming language[26, 27] and the Epigram programming language[21]. These are two examples of recent programming languages that push the application of the Curry-Howard isomorphism down into the realm of "practical" programming. Similarly, the work on the Applied Type System (ATS) [10] looks to incorporate proofs in programs to establish the validity of constraints.

In this paper, we work in Nuprl, [11] a Martin Löf dependent type theory with recursive types and extensional function types. The best and most current account of Nuprl's type theory can be found online [3]. Nuprl might be considered "old" Curry-Howard technology [1], we apply it here to the problem of programming with proofs. This application of Nuprl is not new [25, 15, 6, 7, 9, 8, 18].

1.1 Membership proofs as Indexes

In constructive type theory, proofs implicitly contain programs. In this paper, we apply the propositions-as-types and proofs-as-programs interpretations to examine the structures of the inhabitants of membership predicates over recursively defined types. Under examination, we realize natural generalizations that make these predicates more expressive in terms of the collections of inhabitants they may contain.

Classically, given a structure S of type $\mathcal{S}$ and some element x, a membership predicate $x \in S$ may be true or false. In the constructive setting, if the predicate is true, it is inhabited by the indexes into S leading

*The first authors work was partially supported by NSF grant CCR-9985239.
†This work was supported by a Wyoming NASA Space Grant Graduate Fellowship

[1] Coq [5] and, by now, Lego [19] and Alf[24] are similarly long-in-the-tooth and have been used for similarly rich examples programing via the Curry-Howard isomorphism.

to the element x. For example, if the structure is of type $\mathbb{Z}$ *List* and S is the list $[1;2;2;3;2]$ then, not only is $2 \in S$ true, but its truth is witnessed by the indexes (whose form depends on how the predicate itself is specified) to the second, third and fifth elements of S. When the proposition $2 \in S$ is considered through the propositions-as-types interpretation, it is a type whose elements are the indexes of 2 in S. For tree-like structures, these indexes essentially correspond to paths in the tree. This interpretation of membership predicates as index types arises completely naturally in the constructive setting[2] in the following sense, we prove that the identity function is an isomorphism between a membership predicate and a shape type.

List and tree examples serve to hone intuition and provide a framework for considering some interesting questions related to these ideas in a well understood setting.

In general, we are interested in how the Curry-Howard isomorphism (*i.e.* the propositions-as-types and proofs-as-programs interpretations) can be exploited to explore design spaces. From a methodological view, we show how the identification of membership predicates with index types guides the development of the ideas presented here.

1.2 Related Work

Going back to Jay [17, 16], ideas about shape and polymorphism with respect to it have suggested that the separation of shape from content for recursive types may support generic programming. In that work, Jay proposed viewing recursive types as pairs consisting of a type of indexes together with a list of data elements. Membership functions are not strictly shape polymorphic [23], since indexes to the members depend, not only on the shape of the structure, but on the contents as well.

There has been a recent flurry of interesting work driven from categorical semantics for type theory [1, 4, 2]. These authors are investigating what they call *indexed containers* and what have been called *dependent polynomials* in [13] and polynomial recursive type [12]. We believe that the work described here can be lifted to the more general setting and intend to do so in future work.

We apply constructive type theory to examine the constructive content of membership predicates as indexes.

2 Membership in a Tree

We will motivate the ideas in the context of a type of binary trees storing values of an arbitrary type A in the internal nodes of the tree and whose leaves are empty.

2.1 A binary tree type

We formally define the type of Binary trees as follows:

$$T_A \stackrel{def}{=} \mu X.\, \mathbf{1} + A \times X \times X$$

where $\mu X.\phi$ is a recursive type, the least fixedpoint of type ϕ, which in this case is the polynomial $\mathbf{1} + A \times X \times X$. The disjoint union of types A and B is denoted $A + B$ and the product $A \times B$ denotes Cartesian product of A and B. The type $\mathbf{1}$ is the "unit" type having exactly one element, we call that element "it" and display it as "$\cdot$". For completeness of the propositions-as-types interpretation, every proposition must be interpretable as a type, including equality propositions of the form $x =_A y$. This proposition is meaningful only if both x and y are in A. If true (*i.e.* if x and y are equal in A) the type $x =_A y$ is inhabited by *it*, the single element of the unit type and so is isomorphic to $\mathbf{1}$ and; if false, the type is uninhabited, and is isomorphic to the empty type $\mathbf{0}$.

We have the following well-formedness theorem for T_A.

$$\forall A{:}\mathbb{U}.\, T_A \in \mathbb{U}$$

[2]Indeed we can argue that the only way a membership predicate fails to be an index is if the index information is explicitly discarded in the specification of the membership predicate itself.

It says that for any type, A the type of trees T_A is a type. This type is well-formed because all occurrences of the bound variable X occur in strictly positive positions.

The elements of the type T_A are terms of the form $inl(\cdot)$ (which we will label $Empty$) or are terms of the form $inr(a, t_1, t_2)$ (which we will display as $Node(a, t_1, t_2)$). We remark here that $outl(inl(x)) = x$ and $outr(inr(x)) = x$.

2.2 Membership $x \in_A t$

Membership[3] in a structure of type T_A can be defined most naturally[4] as follows:

Definition 1 (Membership in a tree ($x \in_A t$)).

$$\forall A{:}\mathbb{U}.\ \forall x{:}A.\ \forall t{:}T_A.\ \ (x \in_A t) \in \mathbb{P}$$
$$(x \in_A\ Empty) \stackrel{def}{=} False$$
$$(x \in_A\ Node(y, t_1, t_2)) \stackrel{def}{=} x =_A y\ \vee\ x \in_A t_1\ \vee\ x \in_A t_2$$

The first line gives the well-formedness theorem characterizing the type of the membership predicate. It says that for every type A and for every T_A tree t and every element x of A, $(x \in_A t)$ is a proposition [5]. Note that we are working in a constructive setting and so the disjunction property holds, *i.e.* the proposition $\phi \vee \psi$ holds if at least one of ϕ or ψ holds, *and we know which one.* Thus, the proposition-as-types interpretation indicates that $\phi \vee \psi$ is true if the disjoint union $\phi + \psi$ is inhabited. So, looking back at the definition of membership, inhabitants of $(x \in_A Empty)$ are just the inhabitants of $False$[6] (*i.e.* there are none) and the inhabitants of

$$x =_A y\ \vee\ x \in_A t_1\ \vee\ x \in_A t_2$$

are of the form $inl(u)$ where u inhabits $x =_A y$ or $inr(inl(p))$ where p inhabits the type $(x \in_A t_1)$ or are of the form $inr(inr(p))$ where p inhabits the type $(x \in_A t_2)$. (We assume disjunction associates to the right.)

Thus, inhabitants of this membership predicate (terms that serve as evidence for the truth of an instance of the predicate) are terms in a sum over the types **1** and **0**. The shape of the sum term has a structure matching the shape of the tree where internal nodes are translated as type **1** and leaves are translated as type **0** (since the predicate always returns false on leaves.). *e.g.* the possible inhabitants of the membership in the tree t defined as

$$Node(a, Node(b, Empty, Empty), Node(a, Empty, Empty))$$

are inhabitants of the type

$$\mathbf{1} + ((\mathbf{1} + (\mathbf{0} + \mathbf{0})) + (\mathbf{1} + (\mathbf{0} + \mathbf{0})))$$

To see this, note that the recursive unfolding of the membership predicate $x \in_A t$ evaluates to the following term.

$$x =_A a \vee (((x =_A b) \vee False \vee False) \vee ((x =_A a) \vee False \vee False))$$

Since a occurs twice in this tree, evidence for $a \in t$ takes one of two forms $inl(\cdot)$ or $inr(inr(inl(\cdot)))$ while evidence for $b \in t$ is of the form $inr(inl(inl(\cdot)))$. There are no other inhabitants of this type.

[3] We have rather heavily overloaded the membership symbol. Membership in a type or of a type in a universe should be reasonably easy to distinguish from membership in a tree based on the context. $x \in_A t$ is a defined membership predicate for trees where the type A is a parameter to the predicate indicating which equality to use.

[4] By "naturally" we mean that this is the membership predicate most every functional programmer confronted with this type would write.

[5] $\mathbb{P}_i$ denotes the propositions at some (polymorphic) level i of the hierarchy of propositions. In Nuprl, the hierarchy of propositional universes is just the hierarchy of type universes *i.e.* $\mathbb{P}_i \stackrel{def}{=} \mathbb{U}_i$. We normally write $\mathbb{P}$ ($\mathbb{U}$) for $\mathbb{P}_i$ ($\mathbb{U}_i$).

[6] $False \stackrel{def}{=} \mathbf{0}$

2.3 Abstracting Shape

If t is a tree, we denote the *membershape* of t as $\wr t \int$. We define a mapping from a tree to the sum type representing the shape of the paths to internal nodes as follows:

Definition 2 (member shape).

$$\begin{array}{l} \forall A{:}\mathbb{U}.\, \forall t{:}T_A.\ \ \wr t \int \in \mathbb{U} \\ \wr \mathit{Empty} \int \stackrel{\mathit{def}}{=} \mathbf{0} \\ \wr \mathit{Node}(x, t_1, t_2) \int \stackrel{\mathit{def}}{=} \mathbf{1} + \wr t_1 \int + \wr t_2 \int \end{array}$$

Note that this type is discrete *i.e.* equality on inhabitants of the type $\wr t \int$ is decidable.

Definition 3 (discrete). $\mathit{discrete}\ A \stackrel{\mathit{def}}{=} \forall x, y{:}A.\ x =_A y \vee \neg(x =_A y)$

Thus, a type A is *discrete* if it's equality is decidable. Natural numbers are discrete, the type of functions $\mathbb{N} \to \mathbb{N}$ is not.

Lemma 1 (membershape discrete). $\forall A{:}\mathbb{U}.\, \forall t{:}T_A.\, \mathit{discrete}\ \wr t \int$

A type is finite iff it is in one-to-one correspondence with some initial prefix of the natural numbers.

Definition 4 (finite). $\mathit{finite}\ A \stackrel{\mathit{def}}{=} \exists n{:}\mathbb{N}.\ A \sim \{0 \ldots n\}$

Since trees are defined as least fixedpoints, the type of their indexes is finite.

Lemma 2 (membershape finite). $\forall A{:}\mathbb{U}.\, \forall t{:}T_A.\, \mathit{finite}\ \wr t \int$

The evidence for the finiteness of a type A (a bijection from A to an initial segment of $\mathbb{N}$) gives a method of deciding equality. Thus, all finite types are discrete.

Definition 5 (subtype). $A \subseteq B \stackrel{\mathit{def}}{=} \forall x{:}A.\ x \in B$

Theorem 1. $\forall A{:}\mathbb{U}.\, \forall x{:}A.\ \forall t{:}A\ \mathit{Tree}.\ (x \in_A t) \subseteq \wr t \int$

So, the inhabitants of the membership predicate $x \in_A t$ are inhabitants of the membershape tree for t. To see that these two types are not isomorphic, note that the converse ($\wr t \int \ \subseteq\ (x \in_A t)$) does not hold. Consider a tree where the internal node stores a value (call it y) that is not equal to x, this term in the disjunction will be $y =_T x$ which will be, isomorphic to $\mathbf{0}$, not $\mathbf{1}$. Thus, membershape over approximates the inhabitants of a membership predicate of this form.

If, for some $x \in A$, a tree t, $t \in T_A$ contains the element x at every node, then these types indeed contain the same inhabitants, *i.e.*

$$(x \in_A t) =_{\mathbb{U}} \wr t \int$$

Note that the membershape $\wr t \int$ does not include indexes to the leaves. The definition for the *Empty* case could be modified but then we would loose the close correlation which allows the equality to hold between the inhabitants of the membership predicate $x \in_A t$ and $\wr t \int$ when the nodes of the tree t tree contain only the element x.

2.3.1 Remarks on decidability

Equality on trees is decidable if the underlying type A is discrete.

Lemma 3. $\forall A{:}\mathbb{U}.\, \mathit{discrete}\, A \Leftrightarrow \mathit{discrete}\, T_A$

If the underlying type A is discrete then trees T_A are discrete since equality can be determined by recursively comparing the trees node by node. Perhaps the other direction is slightly less obvious. We reduce the problem of deciding two elements of the underlying type, $x, y \in A$ to that of a decision over the tree type by constructing two trees with the values in question as their node values and comparing the trees for equality:

$$Node(x, Empty, Empty) =_{T_A} Node(y, Empty, Empty)$$

The decision procedure for trees can then, by proxy, decide the equality for A

It should be noted that to actually compute equality of trees (or with the tree-membership predicate $x \in_A t$), there must be a method of deciding equality in the type A. However, even if A is not a discrete type, evidence for $x \in_A t$ takes the form of indexes to nodes in t where values y where $y =_A x$ are stored. Suppose we come by some evidence for $(x \in_A t)$, call this index i. If A is not discrete, we will not be able to verify that the value stored at the node indexed by i actually contains a value equal to x, though the typing tells us it must be. In the context of a proof, such unverifiable evidence is not uncommon, it naturally arises from assumptions specified as antecedents of implications.

2.4 Predicated Shape

We can refine the shape type to take into account the data stored in the nodes of structure as well as its shape, but to do so we need to know the value being searched for. We modify membershape to be sensitive to these issues by generalizing from an individual element of A being searched for in the tree to a predicate that must be satisfied at a node or leaf of the tree.

We write $\wr t \int_\rho$ for the type characterizing the shape of the tree t where the predicate ρ holds.

Definition 6 (predicated member shape).

$\forall A : \mathbb{U}.\ \forall \rho{:}T_A \to \mathbb{P}.\ \forall t{:}T_A.\ \wr t \int_\rho \in \mathbb{U}$

$$\begin{array}{lll} \wr t \int_\rho & \stackrel{def}{=} & \rho(t) \ +\ case\ t\ of \\ & & \quad Empty \ \longrightarrow \mathbf{0} \\ & & \quad \mid Node(x, t_1, t_2) \longrightarrow \wr t_1 \int_\rho \ +\ \wr t_2 \int_\rho \end{array}$$

Note that under the propositions-as-types interpretation, $False$ is defined to be the empty type $\mathbf{0}$. Of course, we could have instead made the predicate $\rho : A \to \mathbb{P}$ but this would not allow indexes to leaves of the structure and we would claim that information is contained both in the indexes of a structure *and* in values stored at internal nodes. Making the predicate over the tree, (*i.e.* of type $T_A \to \mathbb{P}$) allows for more paths in the structure to be indexed, including paths to the empty node.

To see that this definition subsumes shape as given in Def. 2 (which simply depended on the value of the element stored at a node) consider the following predicate which returns true if the node value is x and is false otherwise.

Definition 7 ($x =_A$).

$$\begin{array}{l} \forall A{:}\mathbb{U}.\ \forall x{:}A.\ (x =_A) \in (T_A \to \mathbb{P}) \\ (x =_A Empty) \stackrel{def}{=} False \\ (x =_A Node(y, t_1, t_2)) \stackrel{def}{=} x =_A y \end{array}$$

Using this predicate, we can establish the identity between the inhabitants of the membership predicate $x \in_A t$ and the membershape $\wr t \int_{x=_A}$

Theorem 2. $\forall A{:}\mathbb{U}.\ \forall x{:}A.\ \forall t{:}A\ Tree.\ (x \in_A t) =_{\mathbb{U}} \wr t \int_{x=_A}$

The proof is by induction on the structure of the tree t. It may appear that Nuprl's equality between types is extensional, it is not. Computation in a type (including the unfolding of definitions) does not change it *i.e.* subject reduction is built into Nuprl's type system. Since, by the propositions-as-types interpretation, disjunction ($\vee$) is just defined to be disjoint union ($+$), after a few steps of computation, these types are indeed seen to be intentionally equal.

It is significant that a minor change in the Def. 6 changes the form of the indexes. Consider the following slight alternative to $\wr t \int_\rho$ that we write as $\overline{\wr t \int}_\rho$.

Definition 8 (predicated member shape (alternate)).

$\forall A : \mathbb{U}.\ \forall \rho{:}T_A \to \mathbb{P}.\ \forall t{:}T_A.\ \overline{\wr t \int}_\rho \in \mathbb{U}$

$$\overline{\wr t \int}_\rho \stackrel{def}{=} \rho(t) + \mathit{case}\ t\ \mathit{of}\ \mathit{Empty} \longrightarrow \rho(\mathit{Empty}) \mid \mathit{Node}(x, t_1, t_2) \longrightarrow \overline{\wr t_1 \int}_\rho + \overline{\wr t_2 \int}_\rho$$

The indexes under this slightly modified alternative definition are different from the ones in Def. 6. To see this, assume $\rho(t) = \cdot$ for all trees t. Then $\overline{\wr \mathit{Empty} \int}_\rho$ is inhabited by both $\mathit{inl}(\cdot)$ and $\mathit{inr}(\mathit{inl}(\cdot))$ while the only inhabitant of $\wr \mathit{Empty} \int_\rho$ is $\mathit{inl}(\cdot)$. This would seem to be undesirable in a number of ways. This example illustrates the sensitivity of the evidence on the form of the definition. Clearly, these two shape types are equivalent propositionally, *i.e.* the following holds:

$$\forall A{:}\mathbb{U}.\ \forall \rho{:}T_A \to \mathbb{P}.\ \forall t{:}T_A.\ \ \wr t \int_\rho \Leftrightarrow \overline{\wr t \int}_\rho$$

However, they are not intensionally equal and are not even extensionally equal.

2.5 Predicated Membership

In the same way we have generalized membershape, we can characterize membership in a tree more abstractly by applying a predicate on trees instead of simply searching for a particular element of type A. We continue to abuse notation by writing $\rho \in t$ for the type of indexes to the members of the tree t where ρ holds.

Definition 9 (predicated membership ($\rho \in t$)).

$\forall A{:}\mathbb{U}.\ \forall \rho{:}T_A \to \mathbb{P}.\ \forall t{:}T_A.\ (\rho \in t) \in \mathbb{P}$

$$\rho \in t \stackrel{def}{=} \rho(t) \vee \mathit{case}\ t\ \mathit{of}\ \mathit{Empty} \longrightarrow \mathbf{0} \mid \mathit{Node}(y, t_1, t_2) \longrightarrow \rho \in t_1 \vee \rho \in t_2$$

Now membership and shape are perfectly aligned. This agreement is characterized by the following identity.

Theorem 3. $\forall A{:}\mathbb{U}.\ \forall t{:}T_A.\forall \rho{:}T_A \to \mathbb{P}.\ (\rho \in t) =_{\mathbb{U}} \wr t \int_\rho$

This theorem is easily proved by induction on the structure of the tree t. It says these membership types and membershapes are equal types *i.e.* that they have the same inhabitants and that those inhabitants respect the same equality. Thus, a natural characterization of the type of indexes ($\wr t \int_\rho$) is just the same as predicated membership when we look at it as a type. In the following, we use the two interchangeably.

2.6 Indexing by inhabitants of the membership predicate

Since $\wr t \int_\rho$ is the type inhabited by indexes into t where ρ holds, we should be able to use them as such. The following definition gives a select function; defined so that for each $i \in \wr t \int_\rho$, $t[i]$ is the subtree of t indexed by i.

Definition 10 (select).

$$\begin{array}{l}\forall A{:}\mathbb{U}.\,\forall t{:}T_A.\,\forall \rho{:}T_A\to\mathbb{P}.\,\forall i{:}\lbag t\rbag_\rho.\ t[i]\in T_A\\ t[i] \stackrel{def}{=} \mathit{case}\ i\ \mathit{of}\\ \qquad \mathit{inl}(\_)\to t\\ \qquad \mid \mathit{inr}(y)\to \mathit{case}\ t\ \mathit{of}\\ \qquad\qquad \mathit{Empty}\ \to \mathit{any}(y)\\ \qquad\qquad \mid \mathit{Node}(x,t_1,t_2)\to \mathit{case}\ y\ \mathit{of}\\ \qquad\qquad\qquad \mathit{inl}(\_)\to t_1[\mathit{outl}(\mathit{outr}(i))]\\ \qquad\qquad\qquad \mid \mathit{inr}(\_)\to t_2[\mathit{outr}(\mathit{outr}(i))]\end{array}$$

We note that the term $any(y)$ is computational content of a proof where $\mathbf{0}$ has been assumed; specifically, if $y \in \mathbf{0}$ then, for every type T, $any(y) \in T$. So, *any* maps the paradoxical inhabitant of $\mathbf{0}$ to any type at all. In practice, this behaves like exceptions in ML which take *any* type. An index of the form $inl(\cdots)$ means "This is the indexed node. You're there!". An index of the form $inr(\cdots)$ means, "This is not the node, continue on.". Continue on means, move left or right down the tree and depends on whether $outr(inr(\cdots))$ is of the form $inl(\cdots)$ [go left] or is of the form $inr(\cdots)$ [go right]. So, if the index is of the form $inr(\cdots)$ and the tree is *Empty*, it is an exception to try to continue on – the index extends off the end of the tree and is not well-formed. The well-formedness theorem stipulates that the index i comes from the indexes in the shape $\lbag t\rbag_\rho$ and so this is impossible.

We will write $\lbag t\rbag$ for the shape $\lbag t\rbag_{\lambda x.True}$ which includes *all* indexes into t (including indexes to leaf nodes).

3 Expressiveness of membership

Now, consider the powerset of $\lbag t\rbag$ which we write as $\mathbf{2}^{\lbag t\rbag}$. Since the set $\lbag t\rbag$ is finite, and therefore discrete, the functions in $\mathbf{2}^{\lbag t\rbag} \stackrel{def}{=} \lbag t\rbag \to \mathbf{2}$ are all computable and so this type gives the analog of the classically defined powerset. So, $\mathbf{2}^{\lbag t\rbag}$ is isomorphic to the collection of all subtypes of indexes into the tree t. In some sense we would like know how many of these index sets are expressible using the methods described so far.

Definition 11 (expressible index set). *A type of indexes s where $s \subseteq \lbag t\rbag$ is* expressible *iff there exists a predicate $\rho : T_A \to \mathbb{P}$ such that $s =_{\mathbb{U}} \lbag t\rbag_\rho$.*

With predicated membership ($\rho \in t$) we can, for example, specify indexes of the leaves of t by a predicate that is true when the tree is *Empty* and is false otherwise: $((\lambda x.\, x =_{T_A} Empty) \in t)$. We can specify the collections of all indexes into a tree by the predicate which evaluates to the constant $True$. Similarly, we can express the set of all the internal nodes and many other combinations. However, predicates of type $\rho : T_A \to \mathbb{P}$ are not sensitive to the context or position of a node in a larger tree. So, index sets that are not extensional in ρ (*e.g.* index sets that depend on the context of the indexed node within the tree) are not expressible. For example, no predicate $\rho : T_A \to \mathbb{P}$ can collect the indexes to every other leaf since; no matter where it is encountered, each leaf (*Empty*) is indistinguishable from every other leaf by ρ.

Obviously, predicates $\rho : T_A \to \mathbb{P}$ can not distinguish t from t' if they are equal trees, if $t =_{T_A} t'$ then $\rho\, t =_{\mathbb{P}} \rho\, t'$.

We are after a kind of completeness of expression that the current methods do not give. We have two approaches: (i.) we identify all common subtrees by a quotient construction thereby turning the graph into a directed acyclic graph (DAG) and, (ii.) by extending the predicates to take both the root and an index to the current location thereby gaining a global view of a node in the context of the entire tree.

3.1 Quotienting T_A by common indexes

Since the predicates cannot distinguish equal trees, the indexes so far considered might be quotiented (see [3] and [14]) if they lead to an identical subtree.

Consider the relation which is true if its arguments index equal subtrees.

Definition 12 ($\equiv_t$).

$$\forall A{:}\mathbb{U}.\ \forall t{:}T_A.\ \forall i,j{:}\lfloor t\rfloor.\ \ (i \equiv_t j) \in \mathbb{P}$$
$$i \equiv_t j \stackrel{def}{=} \ t[i] =_{T_A} t[j]$$

This relation is easily seen to be an equivalence relation since type equality on T_A is.

Quotient types (supported in Nuprl) are of the form T/E where T is a type and $E : (T \times T) \to \mathbb{P}$ is an equivalence relation. The inhabitants of the quotient T/E are the equivalence classes on T induced by E. The equivalence classes are named by the elements of the unquotiented type T, and so each notation for an element of T is a notation for an equivalence class in T/E. Each element (equivalence class) in T/E may have many distinct names, but these names all denote the same elements of the quotient type.

In our case, $\lfloor t\rfloor/\equiv_t$ identifies indexes of t if the subtrees they index are equal in the type T_A; equivalently

$$i =_{(\lfloor t\rfloor/\equiv_t)} j \ \text{ iff } \ t[i] =_{T_A} t[j]$$

The quotiented structure indexed in this way is a DAG representation of the tree structure T_A and local predicates are completely expressive with respect to this structure.

By quotienting index sets by $\equiv_t$, we get full expressiveness, in the sense captured by the following theorem.

Theorem 4.

$$\forall A{:}\mathbb{U}.\ \mathit{discrete}\, A \Rightarrow \forall t{:}T_A.\ \forall s{:}\mathbf{2}^{\lfloor t\rfloor}.\ \exists \rho{:}(T_A \to \mathbb{P}).\ (s/\equiv_t) =_{\mathbb{U}} (\lfloor t\rfloor_\rho/\equiv_t)$$

Note that for $s \in \mathbf{2}^{\lfloor t\rfloor}$, we will abuse notation by simply writing s for the indexes in the type $\{i : \lfloor t\rfloor | s(i) = 1\}$, (*e.g.* $s/\equiv_t$ is the type $\{i : \lfloor t\rfloor | s(i) = 1\}/\equiv_t$.)

So, the theorem says that given a tree t over a discrete type A, and given a subset of indexes s, there exists a predicate ρ that perfectly characterizes s modulo the equivalence $\equiv_t$.

The witness for the predicate ρ in the proof is the one that checks if its input (say r) is among the subtrees of t indexed by s. Since this set is finite and since A is discrete, the type T_A is finite and discrete as well, we can just enumerate the trees indexed by s checking if they are equal to r. Now, consider indexes i such that $i \in s$. i is just a name for the equivalence class $\{j \in \lfloor t\rfloor | j \equiv_t i\}$. So the quotient $s/\equiv_t$ (possibly) expands the number of indexes naming its elements. It expands the set of indexes to include those indexes that can not be distinguished by the predicate ρ.

So we see that the quotient essentially grows s to include all the indexes to extensionally equal trees.

3.2 Global Membership Predicates

An alternative to growing the index set s through the quotient construction is to modify the expressiveness of the predicate so that subtrees can be distinguished by their context in the tree being searched. The way to do this is to carry state information through the computation that indicates not only which tree is being evaluated, but its context in the larger tree.

Consider the following dependent type:

$$\forall t{:}T_A.\ \lfloor t\rfloor \to \mathbb{P}$$

Inhabitants of this type have the form $\lambda t.\lambda i.\phi$ where the judgment

$$t{:}T_A,\ i{:}\lfloor t\rfloor \vdash \phi \in \mathbb{P}$$

is derivable. The first argument $t \in T_A$ is the tree being searched and $i \in \lfloor t\rfloor$ is the index to the node currently being examined and ϕ is a proposition determining whether the the tree $t[i]$ is a "member".

The idea for the global search is to pass a dependent predicate (say ρ) of this type both the root of the tree being searched (call it t) and the index i to the current node being searched in the tree. The membership predicate will evaluate $\rho(t)(i)$ and union this result with the search performed on t by appropriately extending i to the left and right branches if t is not the *Empty*. In a way, it is like the cartoon of the train rolling the track out in front of itself.

To implement this plan this we need a way to consistently extend an index.

Definition 13 (Index Composition).

$$\forall A{:}\mathbb{U}.\ \forall t{:}T_A.\ \forall i{:}\wr t\int.\ \forall j{:}\wr t[i]\int.\ \ i \circ j \in \wr t\int$$

$$\begin{array}{lcl} i \circ t & \stackrel{def}{=} & \mathit{case}\ i\ \mathit{of} \\ & & \quad \mathit{inl}(x) \ \rightarrow\ t \\ & & \mid \mathit{inr}(x) \ \rightarrow\ \mathit{case}\ x\ \mathit{of} \\ & & \qquad\quad \mathit{inl}(y) \ \rightarrow\ \mathit{inr}(\mathit{inl}(y \circ t)) \\ & & \qquad \mid \mathit{inr}(z) \ \rightarrow\ \mathit{inr}(\mathit{inr}(z \circ t)) \end{array}$$

The well-formedness goal for the composition is worth studying. It is proved by induction on the structure of the tree t. Note that if t is $Empty$, the only index in $\wr Empty\int$ is $inl(\cdot)$. Since $Empty[inl(\cdot)] = Empty$, the only possible extension of i is j, where $j \in \wr Empty[inl(\cdot)]\int$ which is again $inl(\cdot)$. Looking at the code for composition, the first case says that $inl(\cdot) \circ inl(\cdot) = inl(\cdot)$ which indeed is still an index into $Empty$. The argument in the inductive case is rather straightforward.

The following code implements our strategy for the global search using the composition operator to recursively unroll the indexes down through the tree.

Definition 14 (dependent predicated membership $\rho \in \langle t, i\rangle$).

$$\forall A{:}\mathbb{U}.\ \forall \rho{:}(\forall t{:}T_A.\ \wr t\int \rightarrow \mathbb{P}).\ \forall t{:}T_A.\ \forall i{:}\wr t\int.\ \ (\rho \in \langle t, i\rangle) \in \mathbb{P}$$

$$\begin{array}{lcl} \rho \in \langle t, i\rangle & \stackrel{def}{=} & \rho(t)(i) \vee \mathit{case}\ t[i]\ \mathit{of} \\ & & \qquad \mathit{Empty} \ \rightarrow \mathbf{0} \\ & & \mid \mathit{Node} \rightarrow \rho \in \langle t, i \circ \mathit{inr}(\mathit{inl}(\cdot))\rangle \ \vee\ \rho \in \langle t, i \circ \mathit{inr}(\mathit{inr}(\cdot))\rangle \end{array}$$

That this membership predicate is completely expressive is stated as follows.

Theorem 5.

$$\forall A{:}\mathbb{U}.\ \forall t{:}T_A.\ \forall s{:}\mathbf{2}^{\wr t\int}.\ \exists \rho{:}(\forall t{:}T_A.\ \wr t\int \rightarrow \mathbb{P}).\ \ s =_{\mathbb{U}} (\rho \in \langle t, inl(\cdot)\rangle)$$

To prove it, use the following witness for the existential

$$\lambda t.\ \lambda i.\ \ s(i) = 1$$

Since $s \in \mathbf{2}^{\wr t\int}$, and since $i \in \wr t\int$, $s(i)$ is computable. By this definition $(\lambda t.\ \lambda i.\ \ s(i) = 1)(t)(i)$ will be true whenever the index i is one of the indexes in s and will return $\cdot$ as evidence for that index of s. If $s(i) \neq 1$ then i is not in s.

4 Conclusions and Future Work

We have used the Curry-Howard isomorphism in deep way to identify membership predicates on trees with indexes into those trees.

The issue that came up with the introduction of Def. 8 – regarding the sensitivity of the types of the inhabitants to the formulation of the membership or shape predicate – is potentially a serious issue that needs to be explored more. We could use a quotient type, as we did in Section 3 to mitigate a problem like multiple indexes for one subtree – though we have not done it. In general, the Curry-Howard based evidence is rather delicate and dependent on the language used to generate it. This sensitivity to seemingly small differences in the form an expression may take seems to violate general principles of robustness that is the basis for much software engineering practice. In the end, getting indexes for free from easily specified membership predicates may outweigh any possible drawbacks based on the sensitivity argument. These issues need to be investigated more thoroughly.

We are interested in seeing how the techniques of [20] for data structure transformation might be applied to translations between index types.

As an alternative to the fully expressive membership predicate presented in Def. 14 we are considering formalizing a tree based state monad [29, 22] to pass the global state through the computation.

We do not see any significant technical obstacles to generalizing the results presented in this paper for binary trees to arbitrary polynomial recursive types as presented in [12]. We can imagine extending our abstract data type package for Nuprl to uniformly generate membership and shape types for arbitrary recursive types. Certain that it is related to our efforts here, we are trying to decipher the recent work on indexed containers [4]. Also, although Nuprl's recursive types are least fixedpoints, the index type defined here is not obviously incompatible with coinductive types. We plan to explore this issue in future work.

The authors would like to thank the anonymous referees for their comments.

References

[1] Michael Abbott. *Categories of Containers*. PhD thesis, University fo Leicester, 2003.

[2] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, 2005.

[3] Stuart Allen. *Nuprl Basics*. Cornell University, 2001. `www.cs.cornell.edu/Info/People/sfa/Nuprl/NuprlPrimitives/`.

[4] Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. Indexed containers. Manuscript, available online, February 2006.

[5] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saibi, and Benjamin Werner. *The Coq Proof Assistant Reference Manual : Version 6.1*. Technical Report RT-0203, INRIA, Rocquencourt, France, 1997.

[6] James Caldwell. Moving proofs-as-programs into practice. In *Proceedings, 12th IEEE International Conference Automated Software Engineering*, pages 10–17. IEEE Computer Society, 1997.

[7] James Caldwell. Classical propositional decidability via Nuprl proof extraction. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving In Higer Order Logics*, volume 1479 of *Lecture Notes in Computer Science*, pages 105–122, 1998.

[8] James Caldwell. Extracting recursion operators in Nuprl's type-theory. In A. Pettorossi, editor, *Eleventh International Workshop on Logic -based Program Synthesis, LOPSTR-02*, volume 2372 of *LNCS*, pages 124–131. Springer, 2002.

[9] James Caldwell, Ian Gent, and Judith Underwood. Search algorithms in type theory. *Theoretical Computer Science*, 232(1-2):55–90, Feb. 2000.

[10] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. *SIGPLAN Not.*, 40(9):66–77, 2005.

[11] R. L. Constable et. al. *Implementing Mathematics with the Nuprl proof development system*. Prentice-Hall, 1986.

[12] Marcelo Fiore. Isomorphisms of generic recursive polynomial types. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 77–88, New York, NY, USA, 2004. ACM Press.

[13] Nicola Gambino and Martin Hyland. Wellfounded trees and dependent polynomial functors. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 210–225. Springer, 2003.

[14] Martin Hofmann. *Extensional Constructs in Intensional Type Theory.* CPHC/BCS Distinguished Dissertations. Springer Verlag, London, 1997.

[15] Douglas J. Howe. Reasoning about functional programs in Nuprl. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *Lecture Notes in Computer Science*, Berlin, 1993. Springer Verlag.

[16] C. B. Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.

[17] C. B. Jay and J. R. B. Cockett. Shapely types and shape polymorphism. In D. Sannella, editor, *Programming Languages and Systems (ESOP'94)*, LNCS, pages 302–316. Springer, 1994.

[18] Christoph Kreitz. Building reliable, high-performance networks with the Nuprl proof development system. *Journal of Functional Programming*, 14(1):21–68, 2004.

[19] Z. Luo and R. Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, 1992.

[20] Nicolas Magaud and Yves Bertot. Changing data structures in type theory: A study of natural numbers. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *TYPES*, volume 2277 of *Lecture Notes in Computer Science*, pages 181–196. Springer, 2000.

[21] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[22] Erik Meijer and Johan Jeuring. Merging monads and folds for functional programming. In J. Jeuring and E. Meijer, editors, *Tutorial Text 1st Int. Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, 24–30 May 1995*, volume 925, pages 228–266. Springer-Verlag, Berlin, 1995.

[23] Eugenio Moggi, Gianna Bellè, and C. Barry Jay. Monads, shapely functors, and traversals. *Electr. Notes Theor. Comput. Sci.*, 29, 1999.

[24] Bengt Nordström. The ALF proof editor. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 253–266, Nijmegen, 1993.

[25] James T. Sasaki. *The Extraction and Optimization of Programs from Constructive Proofs.* PhD thesis, Cornell University, 1985.

[26] Tim Sheard. Languages of the future. *SIGPLAN Not.*, 39(12):119–132, 2004.

[27] Tim Sheard. Putting Curry-Howard to work. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 74–85, New York, NY, USA, 2005. ACM Press.

[28] A. Stump. Programming with proofs: Language-based approaches to totally correct software. In N. Shankar, editor, *Verified Software: Theories, Tools, Experiments*, 2005.

[29] Philip Wadler. The essence of functional programming. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, New York, NY, USA, 1992. ACM Press.

Language-Based Program Verification via Expressive Types

Martin Sulzmann and Răzvan Voicu
School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
{sulzmann,razvan}@comp.nus.edu.sg

Abstract

Recent developments in the area of expressive types have the prospect to supply the ordinary programmer with a programming language rich enough to verify complex program properties. Program verification is made possible via tractable type checking. We explore this possibility by considering two specific examples; verifying sortedness and resource usage verification. We show that advanced type error diagnosis methods become essential to assist the user in case of type checking failure. Our results point out new research directions for the development of programming environments in which users can write and verify their programs.

1 Introduction

Program verification is a mature research area, comprising of a wealth of methods and techniques to guarantee useful, sometimes even critical program properties. However, it appears that so far there has not been a good solution for applying program verification techniques to the industrial level software development process. One of the main reasons for this situation is that programmers have to switch to an external tool such as a proof assistant to carry out the often interactive verification process, e.g. consider type-based formal reasoning tools such as Coq [BBC+96], LF [HHP87] and Twelf [PS99]. So far, their integration into a programming language has only reached the "intermediate" level of a programming language [STSP05]. The user cannot directly access them from within her favorite programming language.

On the other hand, automatic type-based checking methods have found its way into todays modern programming languages. By now types form an integral part in the design and implementation of programs and are widely accepted by the user. There are numerous works that show how to capture data invariants via some clever type encodings in languages such as ML [MTM97] and Haskell [Pey03]. In particular, *expressive* type extensions such as type classes with functional dependencies [Jon00] (or variants [CKP05]) and generalized algebraic data types (GADTs) [CH03, XCC03] [1], indicate the possibility of providing the ordinary programmer with a type language rich enough to specify any property of interest.

In this paper, we explore how EADTs (extended algebraic data types) [SWP06a], a previously introduced extension of GADTs with a user-customizable constraint domain, can be used as a systematic method for program verification. There are plenty of other works [CX05, McB, She05, WSW05] that share the same goal. Our work is different in that we apply a combination of advanced type checking *and* type error diagnosis methods to verify (or disprove) complex program properties.

Specifically, our contributions are:

- We explore type error diagnosis methods to assist the user in case of type checking failure (Section 2.3).
- We explore verifying sortedness by going through several trials and (type checking) errors (Section 3).
- We implement a static resource usage verification analysis for a simple while language with EADTs.

[1] GADTs are also known under the name guarded recursive data type or first-class phantom type.

In addition, we apply the type class resolution mechanism to support a mix of static and dynamic verification (Section 4).

In Section 2, we provide background material on EADTs. We conclude in Section 5 where we also discuss related work. We assume some familiarity with type classes [WB89, HHPW96]. Throughout the paper, we will use Haskell-style syntax in examples.

Some of the examples we will see type check using existing implementations such as GHC [GHC]. Others require an experimental language such as Chameleon [SW] or only type check on paper. We expect that in the near future all the examples in this paper are accepted by a revised implementation of Chameleon.

2 Extended Algebraic Data Types

We give a short overview of an extension of Hindley/Milner with user-definable primitive constraints, type annotations and an extended form of algebraic data types (loosely referred to as EADTs). Our focus here is on how to declare EADTs and how to perform type checking [SWP06a]. A description of the semantic meaning of programs and its type soundness proof is given in [SWP06b].

2.1 Extended Data Type Declarations

To illustrate EADTs, we consider the classic append program. Our goal is to express the property that appending two lists yields a list whose length is the sum of the two input lists. Special-purpose systems such as Xi's DependentML [Xi98] or Zenger's index types [Zen99] can easily express such properties. We show that we can mimic such systems via EADTs.

In a first step, we introduce a GADT to model length-indexed lists. We follow the GADT notation as found in GHC [GHC].

```
data Z   = Z    -- singleton types Zero and
data S x = S x  -- Successor
data List a n where
  Nil :: List a Z
  Cons :: a -> List a m -> List a (S m)
```

GADTs are an extension of (boxed) existential types [LO94] and allow to refine the types of constructors via syntactic type equalities. These type assumptions can then be used to type the body of pattern clauses. More precisely, given an expression e of type t we can *change* the type of e to t' if we can verify that under the type assumptions C (i.e. conjunction of type equalities and other primitive constraints such as type classes) types t and t' are equal. We will see examples where we apply such reasonings shortly.

Our next task is to find a suitable encoding of (type) addition. For this purpose, we introduce a ternary (type) constraint symbol `Sum` to represent addition among type level numbers. In the EADT system, the meaning of constraints is user-definable via Constraint Handling Rules (CHR) [Frü95] For example, the following rules encode some common arithmetic laws.

```
rule Sum a b c, Sum a b d ==> c=d          -- (FD)
rule Sum Z a b <==> a=b                    -- (I1)
rule Sum (S a) b d <==> d=S c, Sum a b c   -- (I2)
```

Logically, `==>` denotes Boolean implication and `<==>` denotes Boolean equivalence. Variables in the rule *head* (right-hand side) are universally quantified whereas all remaining variables on the right-hand side are existentially quantified. CHRs also have a simple executable semantics in terms of rewritings among constraints (from left to right). Rules `==>` propagate information whereas rules `<==>` perform simplifications. Examples of CHR derivation steps are given in the next section.

Rule (FD) states that `Sum` must behave like a function. That is, its first two (input) arguments uniquely determine the last (output) argument. Rule (I1) effectively states that `0+a=a` whereas rule (I2) states that

`(a+1)+b=(a+b)+1`. These rules are in fact nothing else but Peano's axioms, and therefore provide for a complete axiomatization of type addition. We will return to this issue in the upcoming Section 2.3.

We glue the existing pieces together and refine the type of `append` as follows.

```
-- EADT encoding of index types
append ::  Sum l m n => List a l -> List a m -> List a n
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

Effectively, we have encoded index types via a combination of GADTs and CHRs. Though, the program text seems to contradict the type annotation. For example, in the first clause we return `ys` which has type `List a m`. This contradicts the annotation which claims that the resulting value has type `List a n`. The important observation is that we need to take into account the "local" constraint `l=Z` arising from the pattern match `Nil` and the constraint `Sum l m n` from the annotation. Together, `Sum l m n` and `l=Z` imply that `m=n` via rule (I1). Hence, we can change the type of `ys` to `List a n`. The critical point is that "local" constraint such as `l=Z` are only visible in their respective branch and therefore do not affect any other program parts. Hence, the first clause is type correct. A similar reasoning applies to the second clause.

In our experience, CHRs are very useful to represent constraint properties. Similar mechanisms have been proposed in the literature [Xi, CX05]. For example, in the Omega language [She] the above CHRs could be specified using the associated types notation [CKP05]

```
type sum :: *->*->*
sum Z m = m
sum (S l) m = S (sum l m)
```

That is, Omega uses a functional rather than relational style of specifying type conditions. We can easily support such source level notations via an appropriate encoding to our internal CHR language (which is Turing complete). Additionally, we can enforce totality by specifying rules such as `rule Sum Int a b ==> False` and other more complex improvement conditions. Something, which is not possible when using a functional style of writing type conditions. Furthermore, via CHRs we can express type classes with extensions such as multi-parameter type classes and functional dependencies [SS05] which are not dealt with in other systems [Xi, She].

2.2 Type Checking

We take a brief look at how to perform type checking for EADTs. We assume that functions are provided with a type annotation. The standard route is to translate the typing problem into a constraint problem. In contrast to standard Hindley/Milner, we need a richer set of *implication* constraints. Such constraints are necessary because we need to type different program parts under different local type assumptions (arising from pattern matches over GADTs and type annotations). For example, in case of

```
append ::  Sum l m n => List a l -> List a m -> List a n
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

we generate the following

$$\forall a, l, m, n. \exists t. \quad (\underline{Sum\ l\ m\ n \wedge t = List\ a\ l \rightarrow List\ a\ m \rightarrow List\ a\ n}) \quad (1) \supset \left(\begin{array}{l} (\underline{\underline{l = Z}} \supset t = List\ a\ l \rightarrow List\ a\ m \rightarrow List\ a\ m) \wedge \\ \forall l'.(\underline{\underline{l = S\ l'}} \supset (Sum\ l'\ m\ n' \wedge t = List\ a\ l \rightarrow List\ a\ m \rightarrow List\ a\ (S\ n')) \end{array} \right)$$

where we use some standard first-order logic notation, e.g. $\wedge$ (Boolean conjunction) and $\supset$ (Boolean implication). The (underlined) right-hand side of the outer implication corresponds to the annotation whereas the (double underlined) right-hand sides of the inner implications correspond to the GADT type equations resulting from the pattern matches.

The above implication constraint can be further simplified by employing the law that $C_1 \supset (C_2 \supset C_3)$ iff $C_1 \wedge C_2 \supset C_3$:

$$\begin{array}{ll} (\forall a,l,m,n.\exists t. & ((Sum\ l\ m\ n, t = List\ a\ l \rightarrow List\ a\ m \rightarrow List\ a\ n \wedge l = Z) \supset \\ & t = List\ a\ l \rightarrow List\ a\ m \rightarrow List\ a\ m)) \qquad \wedge \\ (\forall a,l,m,n.\exists t.\forall l'. & ((Sum\ l\ m\ n, t = List\ a\ l \rightarrow List\ a\ m \rightarrow List\ a\ n \wedge l = S\ l') \supset \\ & (Sum\ l'\ m\ n' \wedge t = List\ a\ l \rightarrow List\ a\ m \rightarrow List\ a\ (S\ n')))) \end{array}$$

In general, the typing problem can expressed as a conjunction of implication constraints of the form $\forall \overline{a_1}.\exists \overline{b_1}...\forall \overline{a_n}.\exists \overline{b_n}.(C_1 \supset C_2)$ where C_1 and C_2 are conjunction of primitive constraints. Universal quantification arises from polymorphic type annotations (e.g. $\forall l,m,n$) and existentially bound data types (e.g. $\forall l'$). Existential quantification arises to represent the type component of type annotations (e.g. $\exists t$) and intermediate variables resulting from Hindley/Milner constraints (we will see such an example at the end of this section).

Our task is to check that under the given CHRs each individual constraint holds (as opposed to type inference where we may need to compute some of the missing assumptions). Roughly, we turn implication checking into an equivalence testing problem by making use of the law that $C_1 \supset C_2$ iff $C_1 \leftrightarrow C_1 \wedge C_2$. We can test whether $C_1 \leftrightarrow C_1 \wedge C_2$ holds by applying CHRs exhaustively on C_1 and $C_1 \wedge C_2$ and check whether the resulting constraints are of the same "canonical" normal form. Of course, we also need to ensure that the quantifiers $\forall \overline{a_1}.\exists \overline{b_1}...\forall \overline{a_n}.\exists \overline{b_n}$ are respected. The exact details of how this process works are given in [SWP06a].

For example, consider the following implication constraint from above.

$$\begin{array}{ll} (\forall a,l,m,n.\exists t. & ((Sum\ l\ m\ n, t = List\ a\ l \rightarrow List\ a\ m \rightarrow List\ a\ n \wedge l = Z) \supset \\ & t = List\ a\ l \rightarrow List\ a\ m \rightarrow List\ a\ m)) \end{array} \quad (1)$$

We find that

$$\begin{array}{ll} & Sum\ l\ m\ n, t = List\ a\ l \rightarrow List\ a\ m \rightarrow List\ a\ n \wedge l = Z \\ \rightarrowtail_{I1} & t = List\ a\ Z \rightarrow List\ a\ m \rightarrow List\ a\ m \wedge l = Z \wedge m = n \quad (2) \end{array}$$

That is, we apply the CHR (I1) from above and normalize the constraint by building the most general unifier. Similarly, we conclude that

$$\begin{array}{ll} & Sum\ l\ m\ n, t = List\ a\ l \rightarrow List\ a\ m \rightarrow List\ a\ n \wedge l = Z \wedge \\ & t = List\ a\ l \rightarrow List\ a\ m \rightarrow List\ a\ m \\ \rightarrowtail_{I1} & t = List\ a\ Z \rightarrow List\ a\ m \rightarrow List\ a\ m \wedge l = Z \wedge m = n \quad (3) \end{array}$$

Hence, final constraints (2) and (3) are equivalent. Hence, statement (1) holds.

This type checking procedure is sound. Decidability follows if we can guarantee that CHRs are terminating. That is, on each (initial) constraint we can only apply a finite number of constraint rewritings. In case of CHR propagation rules we avoid infinite re-propagation by being careful not to apply propagation rules twice on the same constraints. For more details on avoiding re-propagation see e.g. [Abd97].

Completeness depends on two factors. First, we need to guarantee that each constraint has a canonical normal form. This is not necessarily the case for arbitrary CHRs. For example, consider a non-confluent set of CHRs where different rule applications on the same constraint lead to non-joinable constraints. Second, we need to prevent the constraint solver from guessing types. This may happen because some variables are existentially quantified.

Here is a (contrived) example where we make use of multi-parameter type classes [PJM97] which are an instance of the EADT system.

```
class Bar a b where bar :: b->Int
data Foo a where
  Mk :: Bar a b => Foo a
g :: Foo a->Int
g (Mk x) = bar x
```

The type checker is faced with the following implication constraint.

$$\forall a.\exists t.(t = Foo\ a \rightarrow Int \wedge \forall b.(Bar\ a\ b \supset \exists c.Bar\ c\ b))$$

The constraint $\exists c.Bar\ c\ b$ arises out of the program text `bar x` when we build a (generic) type instance for `bar` $:: \forall a, b.Bar\ a\ b \Rightarrow b \rightarrow Int$. The argument type of `x` is bound to a. However, the first parameter in type class Bar is not constrained by any given type. According to Hindley/Milner inference, such types are existentially bound.

In order to verify the above implication constraint we therefore need to guess that c is equivalent to a. Though, guessing the correct type may be non-trivial in general. Type checkers, respectively their underlying constraint solvers, are meant to be deterministic procedures. Hence, we should never have to guess types. In fact, in the type class context the above program is deemed to be illegal. There, guessing types may imply that the programs semantic meaning is ambiguous (e.g. consider [Jon93, SS05] for more details). The above shows that guessing types makes it also hard to obtain a complete type checking procedures. We can rule out such programs by imposing some (conservative) ambiguity checks, thus, rejecting `bar` $:: \forall a, b.Bar\ a\ b \Rightarrow b \rightarrow Int$ because variable a is not constrained by the type $b \rightarrow Int$. Alternatively, we can apply a more liberal strategy and run the implication solver until we reach a situation where we need to guess types. In essence, the type system is then defined by the type checking algorithm.

2.3 Type Error Diagnosis

While it is theoretically interesting to study classes of programs for which type checking is complete, in practice it is much more important to give detailed feedback to the user why a program does not pass the type checker.

Failure of type checking may be due for several reasons. For example, consider the program text `head True` which results in the unsatisfiable constraint $t_h = [a] \rightarrow a \wedge [a] = Bool$. In previous work [SSW03b, SSW04], we have shown how to generate type error messages out of minimal unsatisfiable constraints. These constraints can then be traced back to the specific source locations from which they were generated. Thus, a minimal unsatisfiable constraint represents a possible (minimal) explanation of a type error.

In case of EADTs, failure may arise because we are unable verify that the implication constraint holds. In the example, from above we are unable to verify

$$\forall b.(Bar\ a\ b \supset \exists c.Bar\ c\ b)$$

Effectively, the constraint $Bar\ c\ b$ remains "unmatched". Here, we will take a look at reasons for implication failure in the context of applying EADTs for program verification purposes. We keep the discussion mostly informal. A comprehensive account of the technical aspects underlying our type error diagnosis method is given in [Waz06].

Consider the following variation of an example which we have seen earlier.

```
rule Sum a b c, Sum a b d ==> c=d          -- (FD)
rule Sum Z a b <==> a=b                    -- (I1)
rule Sum (S a) b d <==> d=S c, Sum a b c   -- (I2)
append ::  Sum l m n => List a l -> List a m -> List a n
append Nil ys = Nil -- wrong !
append (Cons x xs) ys = Cons x (append xs ys)
```

Notice that in the first clause we erroneously return `Nil` instead of `ys`. Here is the resulting implication constraint which cannot be verified.

$$\begin{array}{ll}(\forall a, l, m, n.\exists t. & ((Sum\ l\ m\ n, t = List\ a\ l \rightarrow List\ a\ m \rightarrow List\ a\ n \wedge l = Z) \supset \\ & t = List\ a\ l \rightarrow List\ a\ m \rightarrow List\ a\ Z))\end{array}$$

The gist of the problem is that $Sum\ l\ m\ n \wedge l = Z \not\supset n = Z$ for some m (1). We clearly violate the type annotation unless the second list is empty. That is,

$Sum\ l\ m\ n \wedge l = Z \wedge m = Z \supset n = Z$ holds (2)
but $Sum\ l\ m\ n \wedge l = Z \wedge m = S\ m' \not\supset n = Z$ (3).

Based on this information, we can immediately report a static type error. The Chameleon type debugger [SSW03a] will roughly report the following.

```
ERROR: Polymorphic type variable instantiated by
data List a n where Nil ::  List a Z
...
append ::  Sum l m n => List a l -> List a m -> List a n
append Nil ys = Nil
append (Cons x xs) ys = Cons x (append xs ys)
```

The program parts involved in the error are highlighted. This error message provides detailed clues why the program text violates the annotation.

Static failure may also arise in case the CHRs are "incomplete". Here is yet another variation of our running example where we have replaced rule (I1) by (I1').

```
rule Sum a b c, Sum a b d ==> c=d          -- (FD)
rule Sum a Z b <==> a=b                    -- (I1')
rule Sum (S a) b d <==> d=S c, Sum a b c   -- (I2)
append ::  Sum l m n => List a l -> List a m -> List a n
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

The first clause gives rise to the (simplified) implication constraint $Sum\ l\ m\ n \wedge l = Z \supset m = n$ (4). The problem here is that rule (I1') is not suitable to verify statement (4).

A possible fix is to turn (4) into an "axiom". That is, we introduce the additional CHR

```
rule Sum Z m n ==> m=n     -- (I1)
```

Then, the above program text type checks. Of course, we need to ensure that the addition of rule (I2) will not violate the consistency of the resulting set of CHRs. For CHRs we can test for consistency by checking that CHRs are confluent. Confluence guarantees that any order of CHR application will reach the same final constraint. In case of terminating CHRs there is a decidable confluence check [Abd97]. Thus, we can verify that rules (FD), (I1), (I1') and (I2) are confluent. In case we cannot establish termination of CHRs, we cannot apply the mechanical confluence check and must rely on an "external", e.g. human provided, confluence proof.

3 Verifying Sortedness

Our goal is to use the EADT system to express and verify sortedness using the insertion sorting algorithm.

We confine our attention to lists of integers. Integers have singleton types, e.g. `1` has the type `Int(1)` and so on. We assume that the underlying constraint solver of the CHR system is capable of handling primitive constraints `(.<=.)`, `(.<.)` to compare integers on the level of types. [2] The GADT `Test` and the primitive function `test` allows us to "test" the result of comparing two integers.

```
data Test a b = (a<b) => Le | (a>=b) => Gt
test :: Int(a)->Int(b)->Test a b
```

3.1 A First Attempt

We introduce a GADT `SList` to describe sorted lists, in the most straightforward way.

[2] Or, we could add a comprehensive set of CHR rules to specify the meaning of these constraints.

```
-- singleton lists
data N
data C a as
-- sortedness property
rule Sorted N              <==> True
rule Sorted (C a N)        <==> True
rule Sorted (C a (C b bs)) <==>  a<=b, Sorted (C b bs)
-- sorted lists
data SList as where
   Nil :: List N
   Cons :: Int(b) -> SList bs -> SList (C b bs)
```

The insertion sort algorithm is defined in terms of the insertion operation given below.

```
insert :: Int(a)->SList bs->SList cs
insert x Nil = Cons x Nil
insert x (Cons y ys) =                  (1)
   case (test x y) of
      Le -> Cons x (Cons y ys)
      Gt -> Cons y (insert x ys)        (2)
```

The annotation claims that given a sorted list we will return a sorted list. However, the above program will not type check. To understand why let's consider the implication constraint resulting from the last clause (1) and last case (2). We annotate the formula with comments to show which constraints result from which program parts.

$$\forall a, bs, cs.\exists t.\forall b', bs'.$$
$$\left(\begin{array}{l} \left(\begin{array}{l} t = Int(a) \rightarrow Slist\ bs \rightarrow SList\ cs \wedge \\ \quad \texttt{-- insert :: Int(a)->SList bs->SList cs} \\ bs = C\ b'\ bs' \wedge Sorted\ bs \quad \texttt{-- insert x (Cons y ys)} \\ a >= b' \quad \texttt{-- case (test x y) of Gt ->} \end{array}\right) \\ \supset \\ \exists ds, es.\ \ \begin{array}{l} (t = Int(a) \rightarrow SList\ ds \rightarrow SList\ es \wedge \\ es = C\ b'\ ds \wedge Sorted\ es) \end{array} \ \texttt{-- Cons y (insert x ys)} \end{array}\right)$$

The problem becomes now clear. The annotation states that the output list can be any sorted list (even empty list) whereas the constraints from the program text claim that this list is non-empty. Here are the (minimal) constraints involved in this error.

$$\forall cs.\exists t'.\forall b'.t' = SList\ cs \supset \exists ds.t' = SList\ (Cons\ b'\ ds)$$

Clearly, cs is not equal to $Cons\ b'\ ds$ for any cs and b' and some ds. In terms of type error diagnosis, the above could be reported as the type of the output list is too polymorphic.

One way to rectify this error is to constrain `cs` to be non-empty. The signature of `insert` becomes:

```
insert :: Int(a)->SList bs->SList (C c cs)
```

The rest of the program remains the same. However, type checking still fails. Here is the updated implication constraint. Changes are highlighted.

$$
\begin{array}{l}
\forall a, bs, c, cs. \exists t. \forall b', bs'. \\
\left(\begin{array}{l}
\left(\begin{array}{l}
t = \mathit{Int}(a) \rightarrow \mathit{Slist}\ bs \rightarrow \underline{\mathit{SList}\ (C\ c\ cs)} \wedge \\
\quad \texttt{-- insert :: Int(a)->SList bs->SList (C c cs)} \\
bs = C\ b'\ bs' \wedge \mathit{Sorted}\ bs \quad \texttt{-- insert x (Cons y ys)} \\
a >= b' \quad \texttt{-- case (test x y) of Gt ->}
\end{array}\right) \\
\supset \\
\exists ds, e, es. \quad (t = \mathit{Int}(a) \rightarrow \mathit{SList}\ ds \rightarrow \underline{\mathit{SList}\ (C\ e\ es)} \wedge \\
\qquad\qquad \underline{e = b' \wedge \mathit{Sorted}\ (C\ e\ es)} \quad \texttt{-- Cons y (insert x ys)}
\end{array}\right)
\end{array}
$$

The problem is that we yet need to state that the first element of cs is b'.

3.2 A Second Attempt

In the previous example, the sorted list type definition was unable to capture the fact that the first element of the result of `insert` is the minimum between the inserted element `a`, and the first element of array `bs`. We rectify this problem in the new definition of `SList` given below.

```
-- k-sorted lists
data SList k  where
   Nil :: SList k
   Cons :: k<=b => Int(k) -> SList b -> SList k
insert :: Int(a)->SList b ->SList (min a b)
```

The new signature of `insert` states its arguments are an integer and a sorted list whose first element is `b`, and that the result is a non-empty sorted list whose first element is the minimum of `a` and `b`. This program type-checks. To understand how this happens, let us look at the implication constraint resulting from the last clause and case again.

$$
\begin{array}{l}
\forall a, b, bs, cs. \exists t. \\
\left(\begin{array}{l}
\left(\begin{array}{l}
t = \mathit{Int}(a) \rightarrow \mathit{Slist}\ b \rightarrow \mathit{SList}\ (\mathit{min}\ a\ b) \wedge \\
\quad \texttt{-- insert :: Int(a)->SList b->SList (min a b)} \\
a >= b \quad \texttt{-- case (test x y) of Gt ->}
\end{array}\right) \\
\supset \\
(t = \mathit{Int}(a) \rightarrow \mathit{SList}\ b \rightarrow \mathit{SList}\ b \wedge b <= \mathit{min}\ a\ b') \\
\qquad\qquad \texttt{-- Cons y (insert x ys)}
\end{array}\right)
\end{array}
$$

This formula is obviously true. However, the signature of `insert` does not specify its exact behavior; it only states that the output is a sorted list whose first element has a specific property. A stronger, more exact signature, would also specify a set preservation property: the set of elements present in the input arguments is *exactly* the set of elements present in the result returned by `insert`. This property would be captured in the following program.

```
data SList as where
   Nil :: SList N
   Cons :: Sorted (C b bs) => Int(b) -> SList bs -> SList (C b bs)

-- Permutation relation, useful to prove set preservation
rule Permutation N N <=> True
rule Permutation (C a as) bs <=> (Delete a bs zs),(Permutation as zs)
rule Delete a (C a as) as <=> True
rule Delete a (C b bs) (C b cs) <=> Delete a bs cs
```

```
insert :: Permutation (C a as) bs => a -> SList as -> SList bs
-- program text remains the same
```

This program will not type-check. Taking a closer look at the process of applying the type-checking algorithm to this program, we notice that the process stops with the following proof obligations that cannot be further discharged.

$$\begin{aligned} a < b, \mathit{Permutation}\ (C\ a\ (C\ b\ as))\ (C\ b\ bs) &\supset \mathit{Permutation}\ (C\ a\ (b\ as))\ (C\ a\ bs) \\ a \geq b, \mathit{Permutation}\ (C\ a\ as)\ (C\ b\ bs) &\supset \mathit{Permutation}\ as\ bs \end{aligned}$$

While these implication constraints are obviously true, they do not follow directly from the rules of `Permutation`. The above undischarged proof obligations suggest the following alternative rules for `Permutation`.

```
rule Permutation (C a (C b bs)) cs <==>
               a < b | cs = C a ds, Permutation (C b bs) ds -- (**)
rule Permutation (C a (C b bs)) cs <==>
               a >= b | cs = C b ds, Permutation (C a bs) ds -- (**)
```

Here, the guards `a < b` and `a >= b` select exactly one rule for constraint simplification. Indeed, the rules given above define a `Permutation` relation which is satisfied *only if* the two parameters are permutations of each other. However, the following issues need to be addressed here:

- The alternative rules (**) inferred from the failure of type checking, are too restrictive; establishing that they entail a permutation relation (as it is understood in the usual mathematical sense) is not trivial.
- The rules (**) resemble the definition of `insert` very closely. It could be argued that we are rewriting the definition of `insert` at the level of types, which would make the program verification effort superfluous.

This emphasizes the challenges that are faced by program verification via expressive types. In general it is hard to come up with meaningful program properties that are expressed in a way that makes them amenable to automated verification via type-checking for the program at hand. Such properties are usually either not very meaningful (such as a rewrite of the program at the level of types), or too weak to be established. Reconciling these extremes is the subject of further research.

Yet another challenge is coming up with a consistent set of CHR rules that would guarantee that type checking is successful for a comprehensive set of well-typed programs in a specific application domain. In the case of sorting programs, we need a complete theory that is capable of handling inequality constraints, and list permutations. In our examples, we have assumed that such rules were readily available whenever they were needed. Given previous experience with automated proofs of sorting algorithms, a list of rule candidates would be rather straightforward to compile. However, proving that such a system of rules is consistent (or, even better, confluent), complete, and terminating, is subject of further research.

4 Resource Usage Verification

The static verification of the resource usage behavior of programs is a well-studied problem in the literature [Wal00, Thi01]. The task is to ensure that programs conform to a resource usage policy such as a file cannot be accessed after its closed, no read after write etc. Resource usage policies are typically specified via regular expressions. Our goal is to implement such a resource usage analysis with EADTs.

4.1 EADT Implementation

In a first step, we need to predict the run-time resource usage behavior of programs. The standard solution [MSS03] is to abstract a programs resource usage behavior in terms of a type and effect system [TJ92].

Here is a possible EADT implementation for a simple while language where effects are represented by regular expressions.

```
-- Choice
data OR r1 r2
-- Kleene star
data STAR r
-- singleton types/effects
data O -- open
data W -- write
data R -- read
data C -- close
-- EADT to represent type and effect rules
data Cmd r where
  Seq :: Cmd r1 -> Cmd r2 -> Cmd (r1,r2)
  Ite :: Bool -> Cmd r1 -> Cmd r2 -> Cmd (OR r1 r2)
  While :: Bool -> Cmd r1 -> Cmd (STAR r1)
  Open :: Cmd O
  Write :: Cmd W
  Read :: Cmd R
  Close :: Cmd C
```

The constructors encode the typing rules of a type and effect system. For example, in case of if-then-else we use "choice" to express that the resulting effect can be either one of the effects of the two branches. In case of while we use the "Kleene star" to express zero or more number of effects the body of the while statement may have. For each resource primitive we introduce a primitive resource effects. Here are the type and effect rules in a more familiar notation.

$$\text{(Ite)}\ \frac{\vdash exp : Bool \quad \vdash c_1 : Cmd(r_1) \quad \vdash c_2 : Cmd(r_2)}{\vdash If\ exp\ then\ c_1\ else\ c_2 : Cmd(r_1 \mid r_2)}$$

$$\text{(While)}\ \frac{\vdash exp : Bool \quad \vdash c : Cmd(r)}{\vdash While\ exp\ c : Cmd(r^*)} \qquad \text{(Open)}\ \vdash Open : Cmd(O)$$

We can thus analyze that

```
someBoolean :: Bool
prog = Seq Open (Seq (Ite someBoolean Write Read) Close)
```

has type `Cmd (O, ((OR W R),C))`.

To check for resource usage correctness, we need to verify that the effects from the program are a subtype of the effects allowed by the resource usage policy. In our setting, effects are sets of traces where traces are words of a regular language. To implement subtyping among effects, we introduce a binary constraint `Sub` and some appropriate CHRs to program the property that `Sub r1 r2` holds iff $L(r_1) \subseteq L(r_2)$. We spare the readers with the details and refer to [LS04] where we have shown how to implement regular expression language containment using type classes/CHRs.

We consider a specific program on which we impose the policy that after opening a file we may write and read an arbitrary number of times followed by closing the file.

```
check :: Sub r (O, (STAR (OR W R),C)) => Cmd r
prog = check (Seq Open (Seq (Ite someBoolean Write Read) Close))
```

Primitive `check` verifies that the program conforms to the policy.

Here is another example where we impose the policy that after writing to a file we are not allowed to read again.

```
check1 :: Sub r (O, (STAR R, (STAR W, C))) => Cmd r
prog1 = check1 (Seq Open (Seq Write (Seq Read Close))
```

The above program does not type check because the type class constraint `Sub R W` (resulting from the "goal" `Sub (O,(W,(R,C))) (O, (STAR R, (STAR W, C)))` cannot be verified. In Haskell speak, `Sub R W` is an unresolved instance/constraint. This provides us with some important clues where and why our program may violate the resource usage policy.

The next example shows a limitation of our static verification method.

```
check2 :: Sub r W => Cmd r
prog2 :: check2 (Ite True Write Read)
```

The program gives rise to the unresolved type class constraint `Sub W R`. Hence, type checking fails. On the other hand, the program will not violate the resource usage policy because at run-time we will never enter the "then" branch.

The point is that our *static* type checking based verification is incomplete in the sense that some safe programs will be rejected. This is no surprise and the common trade-off for a static analysis to retain decidability. The alternative is to switch to a *dynamic* checking method such as contracts [Mey92] or other forms [Sch00] of monitoring the run-time behavior of programs. Among these two extremes, we seek for a compromise where we attempt to perform as much static checking as possible and turn any unresolved proof obligation into a run-time check.

4.2 Mixing Static and Dynamic Verification

We integrate static with dynamic verification. For this purpose, we make use of the type class resolution mechanism [HHPW96] to instrument programs with run-time checks. The idea is that whenever we come across a program part which gives rise to an unresolved constraint, we replace this program part with an `error` statement (i.e. we raise an exception).

First, we introduce the target language.

```
data Cmd2 r =
   forall r1 r2. (Sub (r1,r2) r) => Seq2 (Cmd2 r1) (Cmd2 r2)
 | forall r1 r2. (Sub (OR r1 r2) r) => Ite2 Bool (Cmd2 r1) (Cmd2 r2)
 | forall r1. (Sub (STAR r1) r) => While2 Bool (Cmd2 r1)
 | (Sub O r) => Open2 | (Sub W r) => Write2
 | (Sub R r) => Read2 | (Sub C r) => Close2
```

Essentially, this is the while language from before. The only difference is that the resource effects are not exact. They can be supersets of the actual effects.

Next, we introduce a set of type class instances to translate an expression of type `Cmd r1` into an expression of type `Cmd2 r2` under the condition that `Sub r1 r2` is satisfied. The translation is driven by the resource type `Cmd r1` of the input program. Recall that types `Cmd r1` are exact. Each `r1` uniquely determines the actual command. Here are the class and instance declarations.

```
class Co r1 r2 | r1-> r2 where coerce :: Cmd r1 -> Cmd2 r2
instance (Co r1 r3, Co r2 r4, Sub (r3,r4) r5)              -- (Seq)
           => Co (r1,r2) r5 where
   coerce (Seq c1 c2) = Seq2 (coerce c1) (coerce c2)
instance (Co r1 r3, Co r2 r4, Sub (OR r3 r4) r5)           -- (Ite)
         => Co (OR r1 r2) r5 where
   coerce (Ite exp c1 c2) = Ite2 exp (coerce c1) (coerce c2)
instance (Co r1 r2,(Sub (STAR r2) r3))                    -- (While)
           => (Co (STAR r1) r3) where
   coerce (While exp c) = While2 exp (coerce c)
instance Sub O r => Co O r where                           -- (Open)
   coerce Open = Open2
instance Sub R r => Co R r where                           -- (Read)
   coerce Read = Read2
instance Sub W r => Co W r where                           -- (Write)
   coerce Write = Write2
instance Sub C r => Co C r where                           -- (Close)
   coerce Close = Close2
```

We convince ourselves that instance (Seq) is correct. The instance context provides `Co r1 r3`, `Co r2 r4` and `Sub (r3,r4) r5`. The pattern match over `Seq (c1 c2)` implies that `r5=(r1,r2)`. The program text `Seq2 (coerce c1) (coerce c2)` gives rise to the constraints `Co r1 r3'`, `Co r2 r4'` and `Sub (r3',r4') r5`. The functional dependency imposed on `Co` guarantees that `r3=r3'` and `r4=r4'`. Hence, the constraints arising are supplied by the instance context. Hence, the instance is correct. Similar observations apply to the other instances.

We now can use the `coerce` method instead of the `check` primitives.

```
prog2 :: Cmd2 W
prog2 = coerce (Ite True Write Read)
```

The program text gives rise to `Co (OR W R) W` which leads to the unresolved constraint `Sub R W`. To support a mix of static and dynamic verification, we fix such failures by introducing additional instances such as

```
instance Co W R where                                      -- (Fail)
  coerce _ = error "run-time error"
instance Sub W R
```

Instances (Write) and (Fail) overlap. Our assumptions is that we first apply the "ordinary" instances. Only in case of failure we will attempt to apply the "fail" instances. In general, for each distinct pair of resource primitives `l1` and `l2` we add declarations

```
instance Co l1 l2 where   coerce _ = error "run-time error"
instance Sub l1 l2
```

Based on the refined type class resolution strategy the type checker accepts `prog2` now. It is illustrative to consider the program after replacing overloaded methods with their actual definitions. The program text of `prog2` gives rise to `Co (OR W R) W`. We need to verify that this constraint holds. Type class resolution proceeds as follows.

Step1: We resolve `Co (OR W R) W` by applying instance (Ite) from above to `Co W W`, `Co W R` and `Sub (OR W R) W`. On the program text level, we rewrite `prog2` to the form

```
prog2' :: Cmd2 W
prog2' :: Ite2 True (coerce Write) (coerce Read)
```

Step2a: We resolve `Co W W` by applying instance (Write) which yields

```
prog2'' :: Cmd2 W
prog2'' :: Ite2 True Write2 (coerce Read)
```

Step2b: We are faced with `Co W R` which previously lead to the unresolved type class constraint error. Now, we can apply instance (Fail) and obtain

```
prog2''' :: Cmd2 W
prog2''' :: Ite2 True Write2 (error "run-time error")
```

In summary, the interesting point here is not so much the fact that we can mix static with dynamic verification. This has already been studied in previous work [CF00, MSS03]. What is interesting here is that we can *implement* a mix of static with dynamic verification via a combination of the type class resolution mechanism and expressive types.

5 Conclusion and Related Work

We have explored to what extent type-based methods are suitable for program verification based on EADTs [SWP06a], our own extension of Hindley/Milner with a user-programmable constraint domain and GADTs. Our results show that besides tractable type checking, improved type error diagnosis methods are highly desirable when verifying complex program properties. The clearly defined EADT type checking procedure makes it possible to provide concise feedback to the user in case type checking fails.

There are a number of related systems such as ATS [Xi, CX05], Cayenne [Aug98], Epigram [McB], Omega [She, She05] and RSP1 [WSW05] which are equally well-suited to support typed-based program verification on the user level. Though, to the best of our knowledge, none of these systems seems to support type error diagnosis.

We have also explored the possibility of mixing the static and dynamic verification of resource usages by using a novel combination of type classes and EADTs. We believe that such an application cannot be expressed in any of the above mentioned systems.

Acknowledgments

We thank the reviewers for their comments.

References

[Abd97] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Proc. of CP'97*, LNCS, pages 252–266. Springer-Verlag, 1997.

[Aug98] L. Augustsson. Cayenne - a language with dependent types. In *Proc. of ICFP'98*, pages 239–250. ACM Press, 1998.

[BBC+96] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliâtre, H. Herbelin, G. Huet, P. Manoury, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual Version 6.1.* INRIA-Rocquencourt-CNRS-ENS Lyon, December 1996.

[CF00] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proc. of POPL'00*, pages 54–66. ACM Press, 2000.

[CH03] J. Cheney and R. Hinze. First-class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, 2003.

[CKP05] M. Chakravarty, G. Keller, and S. Peyton Jones. Associated types synonyms. In *Proc. of ICFP'05*, pages 241–253. ACM Press, 2005.

[CX05] C. Chen and H. Xi. Combining programming with theorem proving. In *Proc. of ICFP'05*, pages 66–77. ACM Press, 2005.

[Frü95] T. Frühwirth. Constraint handling rules. In *Constraint Programming: Basics and Trends*, LNCS. Springer-Verlag, 1995.

[GHC] Glasgow haskell compiler home page. http://www.haskell.org/ghc/.

[HHP87] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. In *Proc. of LICS'87*, pages 194–204. IEEE Computer Society Press, 1987.

[HHPW96] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.

[Jon93] M. P. Jones. Coherence for qualified types. Research Report YALEU/DCS/RR-989, Yale University, Department of Computer Science, September 1993.

[Jon00] M. P. Jones. Type classes with functional dependencies. In *Proc. of ESOP'00*, volume 1782 of *LNCS*. Springer-Verlag, 2000.

[LO94] K. Läufer and M. Odersky. Polymorphic type inference and abstract data types. *ACM Trans. Program. Lang. Syst.*, 16(5):1411–1430, 1994.

[LS04] K. Zhuo Ming Lu and M. Sulzmann. An implementation of subtyping among regular expression types. In *Proc. of APLAS'04*, volume 3302 of *LNCS*, pages 57–73. Springer-Verlag, 2004.

[McB] C. McBride. Epigram: A dependently typed functional programming language. http://www.dur.ac.uk/CARG/epigram/.

[Mey92] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[MSS03] K. Marriott, P. J. Stuckey, and M. Sulzmann. Resource usage verification. In *Proc. of APLAS'03*, volume 2895 of *LNCS*, pages 212–229. Springer-Verlag, 2003.

[MTM97] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, 1997.

[Pey03] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[PJM97] S. Peyton Jones, M. P. Jones, and E. Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, June 1997.

[PS99] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In *CADE*, volume 1632 of *LNCS*, pages 202–206. Springer-Verlag, 1999.

[Sch00] F. B. Schneider. Enforceable security policies. *Information and System Security*, 3(1):30–50, 2000.

[She] T. Sheard. Omega. http://www.cs.pdx.edu/ sheard/Omega/index.html.

[She05] T. Sheard. Putting curry-howard to work. In *Proc. of Haskell'05*, pages 74–85. ACM Press, 2005.

[SS05] P. J. Stuckey and M. Sulzmann. A theory of overloading. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1–54, 2005.

[SSW03a] P. J. Stuckey, M. Sulzmann, and J. Wazny. The Chameleon type debugger. In *Proc. of Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, pages 247–258. Computer Research Repository (http://www.acm.org/corr/), 2003.

[SSW03b] P.J. Stuckey, M. Sulzmann, and J. Wazny. Interactive type debugging in Haskell. In *Proc. of Haskell'03*, pages 72–83. ACM Press, 2003.

[SSW04] P. J. Stuckey, M. Sulzmann, and J. Wazny. Improving type error diagnosis. In *Proc. of Haskell'04*, pages 80–91. ACM Press, 2004.

[STSP05] Z. Shao, V. Trifonov, B. Saha, and N. Papaspyrou. A type system for certified binaries. *ACM Trans. Program. Lang. Syst.*, 27(1):1–45, 2005.

[SW] M. Sulzmann and J. Wazny. Chameleon. http://www.comp.nus.edu.sg/~sulzmann/chameleon.

[SWP06a] M. Sulzmann, J. Wazny, and P.J.Stuckey. A framework for extended algebraic data types. In *Proc. of FLOPS'06*, volume 3945 of *LNCS*, pages 47–64. Springer-Verlag, 2006.

[SWP06b] M. Sulzmann, J. Wazny, and P.J.Stuckey. A framework for extended algebraic data types. Technical report, The National University of Singapore, 2006.

[Thi01] P. Thiemann. Enforcing safety properties using type specialization. In *Proc. of ESOP'01*, volume 2028 of *LNCS*. Springer, April 2001.

[TJ92] J. Talpin and P. Jouvelot. The type and effect discipline. In *Proc. of LICS'92*, pages 162–173. IEEE, 1992.

[Wal00] D. Walker. A type system for expressive security policies. In *Proc. of POPL'00*, pages 254–267. ACM, 2000.

[Waz06] J. Wazny. *Type inference and type error diagnosis for Hindley/Milner with extensions*. PhD thesis, The University of Melbourne, January 2006.

[WB89] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Proc. of POPL'89*, pages 60–76. ACM Press, 1989.

[WSW05] E. Westbrook, A. Stump, and I. Wehrman. A language-based approach to functionally correct imperative programming. In *Proc. of ICFP '05*, pages 268–279. ACM Press, 2005.

[XCC03] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *Proc. of POPL'03*, pages 224–235. ACM Press, 2003.

[Xi] H. Xi. ATS: A language to make typeful programming real and fun. http://www.cs.bu.edu/ hwxi/ATS/ATS.html.

[Xi98] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998.

[Zen99] C. Zenger. *Indizierte Typen*. PhD thesis, Universität Karlsruhe, 1999.

Lightweight static capabilities

Oleg Kiselyov and Chung-chieh Shan
FNMOC and Rutgers University

Abstract

We describe a modular programming style that harnesses modern type systems to verify safety conditions in practical systems. This style has three ingredients:

1. A compact kernel of trust that is specific to the problem domain.
2. Unique names (*capabilities*) that confer rights and certify properties, so as to extend the trust from the kernel to the rest of the application.
3. Static (type) proxies for dynamic values.

We illustrate our approach using examples from the dependent-type literature, but our programs are written in Haskell and OCaml today, so our techniques are compatible with imperative code, native mutable arrays, and general recursion. The three ingredients of this programming style call for (1) an expressive core language, (2) higher-rank polymorphism, and (3) phantom types.

1 Introduction

This paper demonstrates a lightweight notion of *static capabilities* (Walker et al., 2000) that brings together increasingly expressive type systems and increasingly accessible program verification. Like many programmers, we want to assure safety conditions: array indices remain within bounds; modular arithmetic operates on numbers with the same modulus; a file or database handle is used only while open; and so on. The safety conditions protect objects such as arrays, modular numbers, and files. Our overarching view is that a static capability authorizes access to a protected object and simultaneously certifies that a safety condition holds. Rather than proposing a new language or system, our contribution is to substantiate the slogan that types are capabilities, today: we use concrete and straightforward code in Haskell and OCaml to illustrate that a programming language with an appropriately expressive type system is a static capability language.

Section 2 presents a trivial introductory example: eliminating empty-list checking. Section 3 turns to a larger example: eliminating array-bound checking. Section 4 distills the three ingredients of our programming style and describes how expressive it needs the type system to be. Section 5 discusses related and future work.

Our technique scales up: First, shown in the Appendix and online is a more substantial example, Knuth-Morris-Pratt string search. Second, the Takusen database-access project uses our technique to verify the safety of session handles, cursors, prepared-statement handles, and result sets. For instance, any operation on a session is guaranteed to receive a valid session handle. We take our examples from Xi's pioneering work on practical dependent-type systems and Dependent ML, as well as from user suggestions.

2 Eliminating empty-list checking

We start with a trivial example. Although it does not show all features of our approach, it sets the pattern we follow throughout the paper. The example is list reversal with an accumulator, which can be written in OCaml as

```
let rec rev l acc = if null l then acc
                    else rev (tail l) (cons (head l) acc)
```

The code is written for an arbitrary data structure satisfying the list API (`null`, `cons`, `head` and `tail`), so it does not use pattern matching.

The functions `head` and `tail` are partial because they do not make sense for the empty list. Therefore, these functions, to be safe, must check their argument for null before deconstructing it. This code for `rev` checks the same list `l` for null three times: once directly by calling `null`, and twice indirectly in `head` and `tail`.

We can remove excessive checks and gain confidence in the code by prohibiting attempts to deconstruct the empty list. We first define an abstract data type `'a fullList` with the interface and implementation below.

```
module FL : sig
   type 'a fullList
   val unfl : 'a fullList -> 'a list
   val indeed : 'a list -> 'a fullList option
   val head : 'a fullList -> 'a
   val tail : 'a fullList -> 'a list
end = struct
   type 'a fullList = 'a list
   let unfl l = l
   let indeed l = if null l then None else Some l
   let head l = Unsafe.head l
   let tail l = Unsafe.tail l
end
```

Here `Unsafe.head` blindly gives the head of its list argument, without checking if the argument is null. We claim that in well-typed programs the functions `FL.head` and `FL.tail` are total.

Our list reversal with accumulator is now safer and more efficient:

```
let rec rev' l acc = match FL.indeed l with
  | None -> acc
  | Some l -> rev' (FL.tail l) (cons (FL.head l) acc)
```

The code is basically the same as before, but it checks for null only once, as required by the algorithm. The inferred type of `rev'` is the same as that of `rev`, that is, `'a list -> 'a list -> 'a list`.

This example illustrates the basic features of our approach. A *security kernel* `FL` implements an abstract data type `fullList`. Although a `fullList` at run time is the same as a regular `list` and imposes no overhead, it *certifies* a safety condition at compile time, in that a (non-bottom) `fullList` value is never null. The functions `FL.head` and `FL.tail` use this certificate in the type of their argument (rather than a dynamic check) to assure themselves that the access is safe. This certificate is also a *capability* (Miller et al., 2000) that authorizes access to the list elements. This capability is *static* because it is expressed in a type rather than a value (Walker et al., 2000). This idea, to express the result of a dynamic value test as a static type certificate, is important in dependent-type programming (Altenkirch et al., 2005, Section 5). It is reminiscent of safe type-casting in type `dynamic` and of the type-equality assertions of Pašalić et al. (2002).

The functions in the security kernel are generally simple and, as above, non-recursive. In contrast, the *client* code whose safety we eventually wish to assure (`rev` in our example) is recursive. This pattern recurs throughout this paper: in the most complex example, Knuth-Morris-Pratt string search, the client code is imperative and nonprimitively recursive, yet the security kernel just depends on addition, subtraction, and comparison.

Of course, safety depends on the fact that the capability is only issued for a non-empty list. Thus the security kernel has to be verified, perhaps formally. Because `FL.fullList` is opaque, we need only check that `indeed` issues the capability only when the list is non-empty. This claim is straightforward to prove formally:

- On one hand, we could prove along the operational lines of Moggi and Sabry (2001) and Walker et al. (2000) that no expression evaluates to an empty `fullList`. (Such a proof amounts to showing type

safety in an idealized ML-like language whose Hindley-Milner type system is extended with not just list types but also non-empty list types.)

- Or, we could show along the denotational lines of Launchbury and Jones (1995) that the functions in `FL` are parametric even when the logical relation for `fullList` excludes the empty `fullList` (Mitchell and Meyer, 1985).

Either way, our proof is much simpler than these authors' (for example, the logical relation may be unary rather than binary) because our safety condition is much simpler (for example, we do not prove that the `fullList` does not escape some dynamic extent of execution).

The `indeed` function constructs an `option` value that is deconstructed by `rev'` right away. We can eliminate this tagging overhead by changing our code to continuation-passing style.

```
module FLC : sig ...
val indeed : 'a list -> (unit -> 'w) -> ('a fullList -> 'w) -> 'w
let indeed l onn onf = if null l then onn () else onf l
...
let rec revc' l acc = FLC.indeed l (fun () -> acc)
  (fun l -> revc' (FLC.tail l) (cons (FLC.head l) acc))
```

3 Eliminating array-bound checking

We next illustrate our approach on the problem of eliminating unnecessary array-bound checking in binary search (Xi and Pfenning, 1998). This example involves array-index arithmetic and general recursion. All array indexing operations are statically guaranteed to be safe without any run-time overhead. We show OCaml code below; the same idea works in Haskell 98 with higher-rank types. We require minimal type annotations, far simpler than in the dependent-type language. Also, only the small security kernel needs annotations, not the rest of the program.

Below is Xi and Pfenning's original code for the example in Dependent ML (Xi and Pfenning, 1998, Figure 3; see also `http://www-2.cs.cmu.edu/~hwxi/DML/examples/`).

```
datatype 'a answer = NONE | SOME of int * 'a

assert sub <| {n:nat, i:nat | i < n } 'a array(n) * int(i) -> 'a
assert length <| {n:nat} 'a array(n) -> int(n)

fun('a){size:nat}
bsearch cmp (key, arr) =
let
    fun look(lo, hi) =
        if hi >= lo then
            let
                val m = lo + (hi - lo) div 2
                val x = sub(arr, m)
            in
                case cmp(key, x) of LESS => look(lo, m-1)
                                  | EQUAL => (SOME(m, x))
                                  | GREATER => look(m+1, hi)
            end
        else NONE
    where look <| {l:nat, h:int | 0 <= l <= size /\ 0 <= h+1 <= size}
                  int(l) * int(h) -> 'a answer
in
    look (0, length arr - 1)
end
```

```
where bsearch <| ('a * 'a -> order) ->
                 'a * 'a array(size) -> 'a answer
```

The text after <| are dependent-type annotations that the programmer must specify.

3.1 An attempt: parameterized generative modules

This example differs from the one in Section 2 in an important way. There, we merely need to distinguish a non-empty list from a general list, so one abstract type `'a fullList` is enough. Here, to ensure that an array of size n is only accessed with non-negative indices less than n, we need—for each n—one abstract type `'a barray`, for arrays of size n, and another abstract type `bindex`, for non-negative indices less than n. That is, we need two infinite type families, parameterized by the array size n. Because the value n is only known at run-time, dependent types seem called for.

We may try to generate such parameterized abstract types by building a security kernel for each array, that is, by defining a module inside a `let` expression:

```
let bsearch cmp (key, arr) =
  let module BA : sig
   type 'a barray
   type bindex
   val indeed : ('a barray * bindex * bindex) option
   ...
   val bget  : 'a barray -> bindex -> 'a
  end = struct
    type 'a barray = 'a array
    type bindex = int
    let indeed = let n = length arr in
                 if n > 0 then Some (arr, 0, n-1) else None
    let bget = Array.unsafe_get
  end in ...
```

Let us assume that modules are *generative* (see the discussion below).

As in Section 2, we can show that a (non-bottom) value of the type `'a BA.barray` is an array with positive size n, and a (non-bottom) value of the type `BA.bindex` is a non-negative index less than n, where n is the size of the array `arr` in scope for constructing the instance of `module BA`. Consequently, if the expression `BA.get a i` is well typed and `a` and `i` are non-bottom values, then the index `i` is within the bounds of the array `a`.

The safety assurance above relies on generativity: different instances of `module BA` should be type-incompatible. That is, `BA.index` and `BA.barray` must be incompatible types across different instantiations of `module BA`. This generativity corresponds to the "fresh region index" of Moggi and Sabry (2001, Figure 4).

Alas, modules in OCaml are not generative, so the above OCaml code has no guarantees. Modules in SML are generative, but most implementations (including SML/NJ) do not allow constructing a module inside `let`.

3.2 The solution: higher-rank types

Our solution is to emulate the generative module `BA` above using higher-rank types (Mitchell and Plotkin, 1988; Russo, 1998; Shao, 1999a,b; Shields and Peyton Jones, 2001, 2002). The OCaml code below corresponds as closely to the Dependent ML code above as possible. The emulation also works in Haskell, which does not have local module expressions.[1]

[1] Our Haskell code is available online at `http://pobox.com/~oleg/ftp/Haskell/eliminating-array-bound-check-literally.hs`. A slightly more general version at `http://pobox.com/~oleg/ftp/Haskell/eliminating-array-bound-check.lhs` accounts for Haskell arrays with arbitrary lower and upper bounds.

Our solution uses not only higher-rank but also higher-kind types. Rather than using types like `'a barray` and `bindex`, we parameterize them to form types like `('s,'a) barray` and `'s bindex`. We call the extra type parameter `'s` a *brand*.[2] Each possible array size is represented by a type: perhaps `unit` represents 0, `unit list` represents 1, `unit list list` represents 2, and so on. Our kernel use these type-level proxies to brand arrays of that size and indices within that range (Pašalić et al., 2002). We can think of these types as a separate kind `Int`. We do not care which type represents which size; in fact, we use higher-rank polymorphism to generate them arbitrarily.

Suppose that some brand `s` represents some size n. We ensure that a (non-bottom) value of type `(s,'a) barray` is an array of the length n, and a (non-bottom) value of type `s bindex` is a non-negative index less than n. This way, a branded index of the latter type is always in range for a branded array of the former type. We also define types `'s bindexL` and `'s bindexH`, so that a (non-bottom) value of type `s bindexL` is a non-negative index, and a (non-bottom) value of the type `s bindexH` is an index i less than n. (The proof techniques for these invariants are again similar to those mentioned in Section 2. In particular, branding and other kernel operations are parametric under the appropriate unary logical relations (Mitchell and Meyer, 1985): for the type `(s,a) barray`, the set of size-n `a`-arrays, where `s` represents n; for the type `s bindex`, the set of non-negative integers less than n; and so on.)

Our security kernel has the following signature.

```
sig
  type ('s,'a) barray
  type 's bindex
  type 's bindexL
  type 's bindexH

  type ('w,'a) brand_k =
      {bk : 's . ('s,'a) barray * 's bindexL * 's bindexH -> 'w}
  val brand : 'a array -> ('w,'a) brand_k -> 'w

  val bmiddle : 's bindex -> 's bindex -> 's bindex
  val index_cmp : 's bindexL -> 's bindexH
                  -> (unit -> 'w)                    (* if >  *)
                  -> ('s bindex -> 's bindex -> 'w) (* if <= *)
                  -> 'w
  val bsucc : 's bindex -> 's bindexL
  val bpred : 's bindex -> 's bindexH

  val bget  : ('s,'a) barray -> 's bindex -> 'a
  val unbi : 's bindex -> int
end
```

As in Section 2, we use continuation-passing style to avoid tagging overhead. The branding operation `brand` has an essentially higher-rank type: because higher-rank types in OCaml are limited to records, we define a record type `brand_k` with a universally quantified type variable `'s`. Besides branding arrays and indices, the kernel also performs range- (hence, brand-) preserving operations on indices: incrementing an index, decrementing an index, and averaging two indices. The operation `index_cmp l h k1 k2` compares an `indexL` with an `indexH`. If the former does not exceed the latter, we convert both values to `bindex` and pass them to the continuation `k2`. Otherwise, we evaluate the thunk `k1`.

Given such a kernel, we can write the binary search function as follows.

```
let bsearch' cmp (key,(arr,lo,hi)) =
   let rec look lo hi = index_cmp lo hi (fun () -> None)
     (fun lo' hi' ->
       let m = bmiddle lo' hi' in
```

[2] "To burn a distinctive mark into or upon with a hot iron, to indicate quality, ownership, etc., or to mark as infamous (as a convict)." —*The Collaborative International Dictionary of English*

```
        let x = bget arr m in
        let cmp_r  = cmp (key,x) in
        if cmp_r < 0 then look lo (bpred m)
        else if cmp_r = 0 then Some (unbi m, x)
        else look (bsucc m) hi)
    in
    look lo hi

let bsearch cmp (key, arr) =
    brand arr {bk = fun arrb -> bsearch' cmp (key, arrb)}
```

The code follows Xi and Pfenning's Dependent ML code as literally as possible, modulo syntactic differences between SML and OCaml. It is instructive to compare their code with ours. Our algorithm is just as efficient: each iteration involves one middle-index computation, one element comparison, one index comparison, and one index increment or decrement. No type annotation is needed. In contrast, the Dependent ML code requires complex dependent-type annotations, even for internal functions such as `look`. The inferred types for our functions are below.

```
val bsearch' :
  ('a * 'b -> int) ->
  'a *
  (('c, 'b) TrustedKernel.barray * 'c TrustedKernel.bindexL *
   'c TrustedKernel.bindexH) ->
  (int * 'b) option = <fun>
val bsearch :
  ('a * 'b -> int) -> 'a * 'b array -> (int * 'b) option = <fun>
```

To complete the code, we need to implement `TrustedKernel`. The full code is available online;[3] given below are a few notable excerpts. First, we need a way to create values of the type `('s,'a) barray` that ensure that a value of the type `(s,a) barray` is an array of elements of type `a` whose size is represented by the type proxy `s`. Thus we need to generate type proxies for array sizes encountered at run time. McBride (2002) and Kiselyov and Shan (2004) show one such approach in Haskell, which explicitly constructs a type to represent each value. That approach better exposes the connection between branding and dependent types, but it is more general than we need here. We simply generate a fresh type eigenvariable.

```
let brand a k = k.bk (a, 0, Array.length a - 1)
```

The function `bmiddle` is a brand- (that is, range-) preserving operation on branded indices. Its type says that all indices involved have the same brand—that is, the same value range.

```
val bmiddle : 's bindex -> 's bindex -> 's bindex
let bmiddle i1 i2 = i1 + (i2 - i1)/2
```

The type of `bmiddle` corresponds to the proposition

$$0 \le i_1 < n \quad \wedge \quad 0 \le i_2 < n \quad \rightarrow \quad 0 \le \texttt{bmiddle}\ i_1\ i_2 < n,$$

where n is the integer represented by the type proxy `s`. The implementation for `bmiddle` delivers a certificate for the proposition.

```
let index_cmp i j ong onle = if i <= j then onle i j else ong ()
let bsucc = succ and bpred = pred
```

[3] http://pobox.com/~oleg/ftp/ML/eliminating-array-bound-check-literally.ml

3.3 Multiple arrays of various sizes

A more complex example[4] (suggested by a user and a reviewer) is folding over multiple arrays of various sizes. Our goal is a Haskell function

```
marray_fold :: (Ix i, Integral i) =>
         (seed -> [e] -> seed) -> seed -> [Array i e] -> seed
```

which folds over an arbitrary number of arrays, whose lower and upper bounds may differ. The index ranges of some arrays do not even have to overlap and may be empty. Neither the number of arrays to process nor their bounds are statically known, yet we guarantee that all array accesses are within bounds. The key function in this example brands multiple arrays with a type proxy that represents the intersection of their index ranges:

```
brands :: (Ix i, Integral i) => [Array i e]
          -> (forall s. ([BArray s i e], BLow s i, BHi s i) -> w)
          -> w -> w
brands [arr] consumer onempty =
    brand arr (\ (barr,bl,bh) -> consumer ([barr],bl,bh)) onempty
brands (a:arrs) consumer onempty =
    brands arrs (\bbars -> brand_merge bbars a consumer onempty)
                onempty

brand_merge :: (Ix i, Integral i) =>
         ([BArray s i e], BLow s i, BHi s i)
         -> Array i e
         -> (forall s'. ([BArray s' i e], BLow s' i, BHi s' i) -> w)
         -> w -> w
brand_merge (bas,(BLow bl),(BHi bh)) (a :: Array i e) k kempty =
    let (l,h) = bounds a
        l' = max l bl
        h' = min h bh
    in if l' <= h' then
            k (((BArray a)::BArray () i e) :
                (map (\ (BArray a) -> BArray a) bas),
               BLow l', BHi h')
       else kempty
```

Typing this example in a genuinely dependent type system appears quite challenging.

4 Types as static capabilities

In the style just exemplified, the programmer begins verification by building a domain-specific *kernel* module that represents and defends the desired safety condition. This kernel provides *capabilities* to other modules so that they can work safely. Many safety conditions can be expressed using types as proxies for values.

We now describe each step and the language support they need in turn.

4.1 A domain-specific kernel of trust

Program verification typically begins by fixing an assertion language. Given a program, its correctness condition is then extracted automatically or specified manually before being proven. The soundness of the proof checker guarantees that a verified program will behave correctly.

While this approach lets the designer of the verification framework prove soundness once and for all, the desired correctness condition may not reside at the same level of abstraction as the assertion language.

[4] http://pobox.com/~oleg/ftp/Haskell/eliminating-mult-array-bound-check.lhs

Such a mismatch makes the correctness assertion burdensome to construct formally and brittle to prove automatically. For example, if the assertions speak of bytes and registers, then it is hard to verify that modular numbers of different moduli are never mixed together. It is labor-intensive today to translate among layers of representation and verify their correspondence, so this approach works best at a fixed (often low) level of abstraction, as in proof-carrying code (Necula, 1997) and typed assembly language (Morrisett et al., 1999).

We let the programmer design more of the assertion language. For example, it is uncontroversial to let the programmer specify a set of events that need to be checked using temporal logic, rather than fixing a set of events (such as operating-system calls) to track. This way, even given that the framework is sound, whoever uses the framework must ensure that the assertions soundly express the correctness condition desired. In exchange, the programmer can mold the assertion language, for example to express the safety condition for an array index not as a conjunction of inequalities but as an atomic assertion whose meaning is not known to the verifier.

Now that the verification framework no longer knows what the assertions mean, it can no longer build in axioms to justify atomic assertions: because the programmer never defines events in terms of system calls, the framework needs to be told when events occur; because the programmer never defines array bounds in terms of inequalities, the framework needs to be told how to judge an array index in bounds. We call this knowledge a *kernel* of trust, which the programmer creates to represent domain-specific safety conditions.

By extending the kernel of trust, the programmer can verify new safety conditions as needed. Each extension must be scrutinized closely to preserve soundness. In exchange, we gain a "continuum of correctness" in which the programmer can verify more safety conditions as needed.

An expressive programming language allows the user to define and combine a domain-specific library of components. In this regard, the kernel of trust is like any other domain-specific language: its construction relies on succinct facilities for higher-order abstraction.

4.2 Capabilities for extending trust

Our "verifier", the type system, does not track system calls or solve inequalities, but propagates certificates of assertions from the user-defined kernel of trust. Safety then extends from the kernel to the rest of the program. It turns out that type systems are good at this propagation: we trust types.

More precisely, we represent trust by *type eigenvariables*. A type system that supports either higher-rank polymorphism or existential types generates a type eigenvariable fresh in the universal introduction or existential elimination rule (Pierce and Sumii, 2000; Reynolds, 1983; Rossberg, 2003). An opaque type from another module is another instance of a type eigenvariable (Mitchell and Plotkin, 1988). Type eigenvariables are good for representing trust to be propagated, because they are

- unforgeable (so only the kernel of trust can manufacture them),
- opaque (so their identity is the only information they convey), and
- propagated by type inference (so they extend trust from the kernel to the rest of the application).

In other words, type eigenvariables turn a static language of types into a *capability language* (Miller et al., 2000).

The notion of a capability (Miller et al., 2000, Section 3) originated in OS design. A capability is a "protected ability to invoke arbitrary services provided by other processes" (Wulf et al., 1974). For a language system to support capabilities (Miller et al., 2000), access to a particular functionality (for example, access to a collection) must only be via an unforgeable, opaque, and propagated *handle*. For a computation to use a handle, it must have created the handle, received it from another computation, or looked it up in the initial environment. To use a handle, a computation can only propagate it or perform a set of predetermined actions (for example, read an array).

We represent capabilities as types, so we express safety conditions in types, as in dependent-type programming. If a program type-checks, then the type system and the kernel of trust together verify that the safety conditions hold in any run of the program. In most cases, this static assurance costs us no run-time

overhead. In the remaining cases, an optimizing compiler can discover and eliminate statically apparent identity functions at compile time. By guaranteeing safety statically, we can avoid (often excessive) run-time safety checks such as array bound checks.

A capability is commonly viewed as "a pairing of a designated process with a set of services that the process provides" (Miller et al., 2000, Section 3). Hence a special case of a capability, illustrated in Section 2, is an abstract data type. An abstract data type certifies the invariants internal to its implementation: if the implementation preserves the invariants, then the invariants are preserved throughout the application because only the implementation can manipulate values of the abstract type. In general, a capability to access an object certifies the safety condition of that object.

Another example is restricting the IO monad to a few actions. In Haskell, many tasks require the IO monad: file I/O, invoking foreign functions, asking the OS for the time of day or a random number, and so on. The IO monad contains many actions, so a piece of code that can use the IO monad to generate a random number can also use IO to overwrite files on disk and otherwise wreck any guarantee on the code. Instead of providing the code with the IO monad directly, we can provide an monad `m`, where `m` is a type eigenvariable, along with an action of type `m Int` that generates a random integer. Although the program eventually instantiates `m` with the IO monad, the opacity of the type eigenvariable `m` guarantees that the code can only generate random numbers. This basic idea appears in the encapsulation of mutable state by Moggi and Sabry (2001). It is also used realistically in the Zipper file-system project, to statically enforce process separation.

4.3 Static proxies for dynamic values

To express assertions involving run-time values, we associate each value with a type, such that type equality entails value equality. We call these types *proxies* for the values (Pašalić et al., 2002).

The same proxy appearing in the types of multiple values may make additional operations available from the kernel. For example, the branding described in Section 3.2 lets us access an array at an index that is within the bounds of the *same* array. This availability is known as *rights amplification* in the capabilities literature. Miller et al. (2000) writes:

> With rights amplification, the authority accessible from bringing two references together can exceed the sum of authorities provided by each individually. The classic example is the can and the can-opener—only by bringing the two together do we obtain the food in the can.

5 Discussion

We have argued that the Hindley-Milner type system with higher-rank types is a static capability language with rights amplification. Our take on program verification is not to prove the safety conditions from a fixed foundation but to rely on the programmer's trust in a domain-specific kernel. Our technique works in existing languages like Haskell and OCaml, and is compatible with their facilities like mutable cells, native arrays, and general recursion. It requires a modicum of type annotations in the kernel only.

We use types to certify properties of values. For example, the type `s bindex` in Section 3.2 certifies that the index is a non-negative integer less than the array size represented by `s`. The use of an abstract data type whose values can only be produced by a trusted kernel, and the use of a type system to guarantee this last property, is due to Robin Milner in the design of Edinburgh LCF back in the early 1970s (Gordon, 2000). (Incidentally, the language ML—whose early offspring OCaml we use in this paper—was originally designed as a scripting language for the LCF prover.) Our branding technique builds on this fundamental idea using an infinite family of abstract data types, indexed by a type proxy for a run-time value. Our approach still has the serious limitation that we do not produce independently statically checkable certificates.

5.1 On trusting trust

Our lightweight approach depends on a trusted kernel. Because we expect this kernel to vary across applications and change over time, it is harder to trust the kernel, compared to a genuine dependent type system. We have only optimistic speculations to offer at this point.

On one hand, a small kernel may be more amenable to formal treatment than the entire application at once. Even in our most complex examples, verifying imperative and nonprimitively recursive code, our trusted kernel had no recursive functions (and at most relied on simple arithmetic). Seen this way, delineating a kernel of trust is simply a modular strategy towards complete verification. This strategy straddles the line between proof assistants and programming environments, calling for their further integration.

On the other hand, programmers may be more productive, and verification failures more informative, if the framework does not force verifying the part of correctness that is closest to the foundations first. After all, successive refinement of (sketches of) proofs is a time-tested technique. Moving along this "continuum of correctness" may also give a better idea where the code tends to have bugs, and hence where to concentrate verification.

5.2 Dependent type systems

Altenkirch et al. (2005, Section 2) survey dependent type systems and their poor-man emulations. Our use of type proxies and run-time verifiable certificates puts us near the dependently-typed system MetaD (Pašalić et al., 2002). Our work may be thought of as yet another poor man's emulation of a dependent type system (Fridlender and Indrika, 2000; McBride, 2002): instead of putting values in types, we put type proxies for values in types; instead of trusting strongly normalizing terms in type theory, we trust a kernel of uninterpreted capabilities; instead of embracing a new programming practice, we embed in existing programming systems. As Altenkirch et al. write, "many programmers are already finding practical uses for the approximants to dependent types which mainstream functional languages (especially Haskell) admit, by hook or by crook."

Our lightweight approach reasons about the same topics that dependent type systems and optimizing compilers tend to reason about: control flow, aliasing, and ranges. For example, optimizing compilers often perform range analysis to eliminate array bound checking. However, our reasoning kernel is exposed as a module, not tucked away in a compiler and hidden from the view of a regular programmer. Although a genuine dependent type system is preferable in the long run, we are not just exploring a toy: we can express complex reasoning (such as multiple-array bound checking) on real-world applications (such as interfacing to a database) in existing, well-supported language implementations. As mentioned in Section 3.3 above, it is not clear how this expressivity compares with that of today's dependent type systems.

5.3 Mixing static and dynamic checking

Static program analyses are rarely exact because they approximate program behavior without knowing dynamic data. The approximation must be conservative, and so the range analysis, for example, may worry that an index is out of bounds of an array although in reality it is not. To reduce the approximation error, an analysis may insert dynamic checks. Exactly the same is the case for lightweight static capabilities, except the programmer rather than the compiler controls where to insert dynamic checks. We would expect the programmer to understand the program better than the compiler does, and hence to know better where dynamic checks are appropriate and where they are excessive.

A good concrete example is using one index to access two arrays of the same size. Suppose that we want to feed a branded array `ba1` to an array-to-array function `compute_array`, which we expect to return another array `a2` of equal size. We then want to access both arrays using one index. Because `ba1` is branded before `a2` is created, we cannot brand the two arrays at the same time as in Section 3.3. Instead, we can forget the branding of `ba1`, compute the array `a2`, and assign `a2` the brand of `ba1` after a run-time test:

```
let a2 = compute_array (unbrand ba1)
in brand_as a2 ba1 on_mismatched_size (fun ba2 -> ...)
```

The arrays `ba1` and `ba2` now have the same brand. We assume the kernel has generic, application-*independent* functions `unbrand` and `brand_as`.

If we can *prove* that `compute_array` yields an array of size equal to that of its argument, then we can make the function return a branded array, and thus eliminate the run-time size test. Because branding can only be done in the kernel, we must put the function into the kernel, after appropriate rigorous (perhaps formal) verification. The programmer decides whether to expand the trusted kernel for a new application, balancing the cost of the run-time check against the cost of verifying the kernel extension.

The `brand_as` approach is similar to the `assert/cast` dynamic test in MetaD (Pašalić et al., 2002). Such a "cop-out" to deciding type equality is necessary anyway in a dependent type system with general recursion, where type equality is not decidable in general (Altenkirch et al., 2005, Section 3).

5.4 Syntactic sugar

Writing conditionals in continuation-passing-style, as we do here, makes for ungainly code. We also miss pattern matching and deconstructors. These syntactic issues arise because neither OCaml nor Haskell was designed for this kind of programs. The ugliness is far from a show stopper, but an incentive to develop front ends to improve the appearance of lightweight static capabilities in today's programming languages.

Acknowledgments We thank the PLPV reviewers and Mark S. Miller.

References

Altenkirch, Thorsten, Conor McBride, and James McKinna. 2005. Why dependent types matter. Available at `http://www.e-pig.org/downloads/ydtm.pdf`.

Fridlender, Daniel, and Mia Indrika. 2000. Do we need dependent types? *Journal of Functional Programming* 10(4):409–415.

Gordon, Michael J. C. 2000. From LCF to HOL: a short history. In *Proof, language, and interaction*, ed. G. Plotkin, Colin P. Stirling, and Mads Tofte. MIT Press.

Kiselyov, Oleg, and Chung-chieh Shan. 2004. Functional pearl: Implicit configurations—or, type classes reflect the value of types. In *Proceedings of the 2004 Haskell workshop*. New York: ACM Press.

Launchbury, John, and Simon L. Peyton Jones. 1995. State in Haskell. *Lisp and Symbolic Computation* 8(4):293–341.

McBride, Conor. 2002. Faking it: Simulating dependent types in Haskell. *Journal of Functional Programming* 12(4–5):375–392.

Miller, Mark S., Chip Morningstar, and Bill Frantz. 2000. Capability-based financial instruments. In *Financial cryptography*, ed. Yair Frankel, vol. 1962 of *Lecture Notes in Computer Science*, 349–378. Springer.

Mitchell, John C., and Albert R. Meyer. 1985. Second-order logical relations (extended abstract). In *Logics of programs*, ed. Rohit Parikh, 225–236. Lecture Notes in Computer Science 193, Berlin: Springer-Verlag.

Mitchell, John C., and Gordon D. Plotkin. 1988. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems* 10(3):470–502.

Moggi, Eugenio, and Amr Sabry. 2001. Monadic encapsulation of effects: A revised approach (extended version). *Journal of Functional Programming* 11(6):591–627.

Morrisett, Greg, David Walker, Karl Crary, and Neal Glew. 1999. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems* 21(3):527–568.

Necula, George C. 1997. Proof-carrying code. In *POPL '97: Conference record of the annual ACM symposium on principles of programming languages*, 106–119. New York: ACM Press.

Pašalić, Emir, Walid Taha, and Tim Sheard. 2002. Tagless staged interpreters for typed languages. In *ICFP '02: Proceedings of the ACM international conference on functional programming*, 218–229. New York: ACM Press.

Pierce, Benjamin, and Eijiro Sumii. 2000. Relating cryptography and polymorphism. Available at `http://www.yl.is.s.u-tokyo.ac.jp/~sumii/pub/`.

Reynolds, John C. 1983. Types, abstraction and parametric polymorphism. In *Proceedings of 9th IFIP world computer congress, Information Processing '83*, ed. R. E. A. Mason, 513–523. Amsterdam: North-Holland.

Rossberg, Andreas. 2003. Generativity and dynamic opacity for abstract types (extended version). Tech. Rep., Universität Saarbrücken. Also PPDP2003.

Russo, Claudio V. 1998. Types for modules. Ph.D. thesis, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh. Also as Tech. Rep. ECS-LFCS-98-389.

Shao, Zhong. 1999a. Transparent modules with fully syntactic signatures. In *ICFP '99: Proceedings of the ACM international conference on functional programming*, vol. 34(9) of *ACM SIGPLAN Notices*, 220–232. New York: ACM Press.

———. 1999b. Transparent modules with fully syntactic signatures. Tech. Rep. YALEU/DCS/TR-1181, Department of Computer Science, Yale University, New Haven.

Shields, Mark B., and Simon L. Peyton Jones. 2001. First-class modules for Haskell. Tech. Rep., Microsoft Research. `http://www.cse.ogi.edu/~mbs/pub/first_class_modules/`.

———. 2002. First-class modules for Haskell. In *9th international workshop on foundations of object-oriented languages*, 28–40. New York: ACM Press.

Walker, David, Karl Crary, and Greg Morrisett. 2000. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems* 22(4):701–771.

Wulf, William A., Ellis S. Cohen, William M. Corwin, Anita K. Jones, Roy Levin, C. Pierson, and Fred J. Pollack. 1974. Hydra: The kernel of a multiprocessor operating system. *Commun. ACM* 17(6):337–345.

Xi, Hongwei, and Frank Pfenning. 1998. Eliminating array bound checking through dependent types. In *PLDI '98: Proceedings of the ACM conference on programming language design and implementation*, vol. 33(5) of *ACM SIGPLAN Notices*, 249–257. New York: ACM Press.

Appendix: Knuth-Morris-Pratt string matching in Dependent ML and Haskell

We borrow another involved example from Xi and Pfenning (1998), Knuth-Morris-Pratt string matching (KMP). This algorithm uses mutable arrays whose elements' values determine indices in turn. It also uses the deliberately one-off index -1 as a special flag. Our Haskell code[5] using higher-rank types has the same run-time costs and static guarantees as the Xi and Pfenning's Dependent ML code: all array and string operations are verified to be safe.

[5] `http://pobox.com/~oleg/ftp/Haskell/KMP-deptype.hs`

Statically Verified Type-Preserving Code Transformations in Haskell

Louis-Julien Guillemette Stefan Monnier
Université de Montréal
{guillelj,monnier}@iro.umontreal.ca

Abstract

The use of typed intermediate languages can significantly increase the reliability of a compiler. By type-checking the code produced at each transformation stage, one can identify bugs in the compiler that would otherwise be much harder to find. We propose to take the use of types in compilation a step further by verifying that the transformation itself is type correct, in the sense that it is impossible that it produces an ill typed term given a well typed term as input.

We base our approach on higher-order abstract syntax (HOAS), a representation of programs where variables in the object language are represented by meta-variables. We use a representation that accounts for the object language's type system using generalized algebraic data types (GADTs). In this way, the full binding and type structure of the object language is exposed to the host language's type system. In this setting we encode a type preservation property of a CPS conversion in Haskell's type system, using witnesses of a type correctness proof encoded as values in a GADT.

1 Introduction

While there is still a long way to go until they become as common place as in digital systems, formal methods are rapidly improving and gaining ground in software. Type systems are arguably the most successful and popular formal method used to develop software, even more so since the rise of Java. For this reason, there is a lot of interest in trying to beef up type systems incrementally to enable them to prove more complex properties.

Thus as the technology of type systems progresses, new needs and new opportunities appear. One of those needs is to ensure the faithfulness of the translation from source code to machine code. After all, why bother proving any property of our source code, if our compiler can turn it into some unrelated machine code? One of the opportunities is to use types to address this need. This is what we are trying to do.

Typed intermediate languages have been used in compilers for various purposes such as type-directed optimization [8, 23, 15], sanity checks to help catch compiler errors, and more recently to help construct proofs that the generated code verifies some properties [11, 6]. Typically the source level types are represented in those typed representations in the form of data-structures which have to be carefully manipulated to keep them in sync with the code they annotate as this code progresses through the various stages of compilation. This has several drawbacks:

- Additional work, obviously, which can slow down our compiler. To minimize the impact, the type language and the type annotations have to be very carefully designed and coded, using techniques like hash-consing, explicit substitutions, and other optimizations [18].
- Occasionally, the need to update the type annotations can make an optimization impractical, e.g. because the necessary type information is not immediately available and thus requires restructuring the algorithm.

- Need to choose between different design tradeoffs: either place only as few type annotations as possible to reduce the impact of the first problem above, or on the contrary, add type annotations everywhere to reduce the risk of bumping into the second problem above.
- Errors are only detected when we run the type checker, but running it as often as possible slows down our compiler even more.
- This amounts to *testing* our compiler, thus bugs can lurk, undetected.

To avoid those problems, we want to represent the source types of our typed intermediate language as types instead of data. This way the type checker of the language in which we write our compiler can *verify* once and for all that our compiler preserves the typing correctly. The compiler itself can then run at full speed without having to manipulate and check any more types. Also this gives us even earlier detection of errors introduced by an incorrect program transformation, and at a very fine grain, since it amounts to running the type checker after every instruction rather than only between phases.

The type-preservation argument has been introduced into the implementation of a compiler using a typeful program representation in [2]. But to our knowledge, the work presented here is the first attempt to formally establish a type preservation property using a language so widely used and well supported as Haskell, for which a industrial strenth compiler is available.

This work follows a similar goal to the one of [9], but we only try to prove the correctness of our compiler w.r.t the static semantics rather than the full dynamic semantics. In return we want to use a more practical programming language and hope to limit our annotations to a minimum such that the bulk of the code should deal with the compilation rather than its proof. Also we have started this work from the frontend and are making our way towards the backend, whereas Leroy's work has started with the backend. Our contributions are the following:

- We show a type-preserving CPS translation written in Haskell and where the GHC compiler verifies the property of type-preservation.
- We extend the classical toy example of a generalized algebraic data type (GADT) representation of an abstract syntax tree, to a full language with bindings.
- We use higher-order abstract syntax (HOAS) in our intermediate representation, following [24], and we show how to combine this technique with GADTs and how to build such terms using Template Haskell.

The remainder of this paper is organized as follows. We review generalized algebraic datatypes and the notion of higher-order abstract syntax in Sec. 2. Section 3 presents the CPS conversion, states a type-preservation property that it satisfies, and then shows how we encoded it in Haskell. Section 4 presents some alternative approaches, as well as some solutions to some of the problems we encountered. Section 5 mentions related work and Sec. 6 concludes.

2 Background

In this section we develop a typeful program representation using GADTs and higher-order abstract syntax for a simple source language that is a simply-typed λ-calculus with pairs and integers (herein called $\lambda_{\rightarrow}$.) We briefly describe the programming techniques used for manipulating such a representation based on Washburn and Weirich's work [24].

2.1 Generalized algebraic datatypes

Generalized algebraic datatypes (GADTs) [25, 3] are a generalization of algebraic datatypes where the return types of the various data constructors for a given datatype need not be identical – they can differ in the type arguments given to the type constructor being defined. The type arguments can be used to encode

$$
\begin{array}{ll}
(\textit{types}) & \tau ::= \tau_1 \to \tau_2 \mid \mathsf{int} \mid \tau_1 \times \tau_2 \\
(\textit{type env}) & \Gamma ::= \bullet \mid \Gamma, x{:}\tau \\
(\textit{primops}) & p ::= + \mid - \mid \times \\
(\textit{exps}) & e ::= x \mid \lambda x{:}\tau_1.\, e : \tau_2 \mid e_1\ e_2 \mid (e_1, e_2) \mid \pi_i\ e \mid i \mid e_1\ p\ e_2 \\
 & \quad \mid \mathsf{if0}\ e_1\ e_2\ e_3
\end{array}
$$

Typing rules

$$
\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\, e : \tau_2 : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}
$$

$$
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i\ e : \tau_i} \qquad \frac{}{\Gamma \vdash i : \mathsf{int}}
$$

$$
\frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1\ p\ e_2 : \mathsf{int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{if0}\ e_1\ e_2\ e_3 : \tau}
$$

Figure 1: $\lambda_{\to}$ syntax and static semantics

additional information about the value that is represented. For our purpose, we use GADTs to represent abstract syntax trees, and use these type annotations to track the source-level type of an expression.

Consider the language $\lambda_{\to}$ defined in Fig. 1. The fragment of $\lambda_{\to}$ concerned with integers could be represented in a GADT as follows:

```
data Exp t where
  Num   :: Int                            -> Exp Int
  Prim  :: PrimOp -> Exp Int -> Exp Int -> Exp Int
  If0   :: Exp Int -> Exp t -> Exp t     -> Exp t

data PrimOp = Add | Sub | Mult
```

This `Exp` data type not only defines the abstract syntax but also encodes the typing rules of our language. E.g. a statement such as $\Gamma \vdash e : \tau$ is represented in Haskell by the fact that `e` :: `Exp` τ. The environment Γ is kept implicit. Note also that an expression of type `Exp t` represents a $\lambda_{\to}$ expression of *source type* t, where we have (arbitrarily) chosen the Haskell type `t` to stand for the corresponding $\lambda_{\to}$ type t (e.g. we use the Haskell type `Int` to represent the $\lambda_{\to}$ type int.)

Extending this encoding for the variable and the λ cases is not straightforward since we kept Γ implicit. Of course, we could try to make Γ explicit as in `Exp t` Γ, but that can quickly become cumbersome since it can entail reifying variables at the level of types, and encoding structural rules such as weakening and exchange. So instead, we use higher-order abstract syntax which allows us to keep Γ implicit.

2.2 Higher-order abstract syntax

Higher-order abstract syntax (HOAS) [14] is a program representation where variables in the object language are represented using meta-variables. For instance, functions in our source language would be represented using Haskell functions; thus we could extend the representation of the language to account for $\lambda_{\to}$ functions as follows:

```
data Exp t where
  ...
  Lam   :: (Exp s -> Exp t)              -> Exp (s -> t)
  App   :: Exp (s -> t) -> Exp s         -> Exp t
```

As is apparent from this definition, the typing rule for functions in $\lambda_{\to}$ can be expressed straightforwardly in terms of Haskell's typing rule for functions.

```
data ExpF a where
  Lam   :: (a t1 -> a t2)           -> ExpF (a (t1 -> t2))
  App   :: a (t1 -> t2) -> a t1     -> ExpF (a t2)

  Pair  :: a t1 -> a t2             -> Expf (a (t1, t2))
  Fst   :: a (t1, t2)               -> ExpF (a t1)
  Snd   :: a (t1, t2)               -> ExpF (a t2)

  Num   :: Int                      -> ExpF (a Int)
  Prim  :: PrimOp -> a Int -> a Int -> ExpF (a Int)
  If0   :: a Int -> a t -> a t      -> ExpF (a t)

data Rec a b t = Roll (a (Rec a b t))  | Place (b t)

type Exp a t = Rec ExpF a t
```

Figure 2: Typeful, parametric representation of $\lambda_{\rightarrow}$

It is difficult in general to define recursive functions over higher-order terms. The problem comes from the fact that, in order to inspect the term "under a binder", one has to apply the corresponding meta-level function – and then, what information must be passed as argument? To alleviate this difficulty, it is useful to make use of an elimination form, commonly called a catamorphism (or iterator; we use the two terms interchangeably here, although they are given more specific meaning elsewhere [24]). A catamorphism encapsulates the traversal of a recursive structure; more precisely, it is a (higher-order) function that, given an elementary operation to perform on a single element, applies this operation to every element of the structure. (The most familiar instance of a catamorphism being the `fold` function over lists.)

In this work, we make use of Fegara and Sheard's catamorphism [5], over a parametric program representation encoded in Haskell [24]. In the remainder of this section, we briefly show how such an iterator is used and what modifications must be made to the (naive) program representation shown above, and illustrate its use with a simple example.

Figure 2 shows the representation we use. It differs from the naive representation in two ways:

1. It has been split into two types, `ExpF` and `Exp`, in order to make the recursive structure of the representation explicit. The type `ExpF` is the "prototype" representation, where the type argument `a` stands for the recursive form of the type. The recursive form, defined by the type `Exp`, is obtained by application of a sort of fixed-point operator, which is represented by the type constructor `Rec`. (You can ignore the data constructor `Place`, used internally by the iterator; see [5] if you are curious).

2. The representation is *parametric* in a type argument `a`, that is, a $\lambda_{\rightarrow}$ term of source type t is represented by a term of type `∀a. Exp a t`, where `t` is the Haskell type that represents t. When applying the catamorphism, the type variable `a` is instantiated with the type that represents the information associated with a term (for instance, in the example of the the pretty-printer below, that information is the textual representation of the term represented as a string.)

Figure 3 shows the type of the iterator along with an example of its application. The internal functions `xmapExpF`, `cata`, and `iter` are taken from [24] and adapted to the case of a typed representation. The pretty-printer implementation consists of two functions: `showAux`, which shows an individual node of the syntax tree, and `showE`, which shows an entire tree and is obtained by application of the iterator.

Indeed, in our higher-order program representation, program variables are represented as Hasell variables, and thus have no identifiers associated with them. The pretty-printer assigns identifiers to variables as the traversal proceeds. The information associated with a term is its textual representation, which is parameterized by a list of identifiers; thus the type of terms `∀a. Exp a t` is instantiated as `Exp ([String] -> String) t`. In an imperative language, we would have simply used a *gensym* facility, but Haskell being side-effect-free, we have to thread a list of available identifiers in the display function.

```
xmapExpF :: (∀t. (a t -> b t, b t -> a t))
         -> (∀t. (ExpF (a t) -> ExpF (b t), ExpF (b t) -> ExpF (a t)))
cata  ::  (∀t. (ExpF (a t) -> a t)) -> (∀t. Exp a t -> a t)
iter  ::  (∀t. ExpF (b t) -> b t) -> (∀t. ((∀a. Exp a t) -> b t))

showAux :: ExpF ([String] -> String) -> ([String] -> String)
showAux (Num n) (v:vars) = show n
showAux (App x y) vars =
 "(" ++ (x vars) ++ " " ++ (y vars) ++ ")"
showAux (Lam z) (v:vars) =
 "(fn " ++ v ++ "= " ++ (z (const v) vars) ++ ")"
...

showE :: (∀a. Exp a t) -> String
showE e = iter showAux e vars
  where vars = ['a' .. 'z'] ++ ...
```

Figure 3: Pretty-printer implementation using Fegara and Sheard's iterator.

3 CPS conversion

In this section we present the core contribution of this paper: an implementation of a CPS transformation where the type system of Haskell is used to encode the proof that this implementation correctly preserves types.

We proceed as follows. We first show the CPS conversion in its theoretical form; then define the typed representation of the target language λ_K; then show how to encode witnesses of type correspondence using existential types and GADTs; and finally show how the *functional dependency* between a type and its CPS form, a crucial point for completing the type correspondence proof, can also be encoded using GADTs.

3.1 The theory

Conversion to continuation-passing style (CPS) names all intermediate computational results and makes the control structure of a program explicit. In CPS, a function does not return a value to the caller, but instead communicates its result by applying a *continuation*, which is a function that represents the "rest of the program", that is, the context of the computation that will consume the value produced. The target language of the CPS conversion, here called λ_K has the following syntax:

$$
\begin{array}{ll}
(\textit{types}) & \tau ::= \tau \to 0 \mid \mathsf{int} \mid \tau_1 \times \tau_2 \\
(\textit{type env}) & \Gamma ::= \bullet \mid \Gamma, x{:}\tau \\
(\textit{values}) & v ::= x \mid i \mid \lambda x{:}\tau\ .\ e \mid (v_1, v_2) \\
(\textit{primops}) & p ::= + \mid - \mid \times \\
(\textit{exps}) & e ::= \mathsf{let}\ x = v\ \mathsf{in}\ e \mid \mathsf{let}\ x = \pi_i\ v\ \mathsf{in}\ e \mid \mathsf{let}\ x = v_1\ p\ v_2\ \mathsf{in}\ e \\
 & \qquad \mid v_1\ v_2 \mid \mathsf{if0}\ v\ e_1\ e_2 \mid \mathsf{halt}\ v
\end{array}
$$

It differs from $\lambda_\to$ in that its syntax is split into two syntactic categories of expressions and values. Values represent those things that can be bound to a variable: either another variable, or the introduction forms for functions, integers or pairs. Expressions consist of a list of declarations (introduced by let forms), followed by either a function application, a conditional expression, or the special form halt, which indicates the final "answer" produced by the program. The fact that a function does not return to the caller is reflected in its type as $\tau \to 0$.

Figure 4 shows the CPS conversion itself. It is defined in three functions. The main function, $\mathcal{K}[\![-]\!]\ \kappa$, transforms a $\lambda_\to$ expression in its CPS form expression, given a continuation κ. The function $\mathcal{K}_{\text{type}}[\![-]\!]$, for each type in $\lambda_\to$, gives the corresponding type in λ_K. (Note that this function is used to convert the type annotations in the case $\mathcal{K}[\![\lambda x{:}\tau_1.\, e : \tau_2]\!]$). Finally, $\mathcal{K}_{\text{prog}}[\![-]\!]$ converts an entire program by arranging for the final result to be passed to the special form halt.

$$
\begin{array}{rcl}
\mathcal{K}_{\text{type}}[\![\mathsf{int}]\!] & = & \mathsf{int} \\
\mathcal{K}_{\text{type}}[\![\tau_1 \times \tau_2]\!] & = & \mathcal{K}_{\text{type}}[\![\tau_1]\!] \times \mathcal{K}_{\text{type}}[\![\tau_2]\!] \\
\mathcal{K}_{\text{type}}[\![\tau_1 \to \tau_2]\!] & = & (\tau_1 \times (\tau_2 \to 0)) \to 0 \\
\\
\mathcal{K}[\![x]\!]\ \kappa & = & \kappa\ x \\
\mathcal{K}[\![\lambda x{:}\tau_1.\, e : \tau_2]\!]\ \kappa & = & \kappa\ (\lambda(x,c){:}\mathcal{K}_{\text{type}}[\![\tau_1]\!] \times (\mathcal{K}_{\text{type}}[\![\tau_2]\!] \to 0).\, \mathcal{K}[\![e]\!]\ \ c) \\
\mathcal{K}[\![e_1\ e_2]\!]\ \kappa & = & \mathcal{K}[\![e_1]\!]\ (\lambda x_1.\ \mathcal{K}[\![e_2]\!]\ (\lambda x_2.\ x_1\ (x_2, \kappa))) \\
\mathcal{K}[\![(e_1, e_2)]\!]\ \kappa & = & \mathcal{K}[\![e_1]\!]\ (\lambda x_1.\ \mathcal{K}[\![e_2]\!]\ (\lambda x_2.\ \kappa\ (x_1, x_2))) \\
\mathcal{K}[\![\pi_i\ e]\!]\ \kappa & = & \mathcal{K}[\![e]\!]\ (\lambda x.\ \mathsf{let}\ x = \pi_i\ x\ \mathsf{in}\ \kappa\ x) \\
\mathcal{K}[\![i]\!]\ \kappa & = & \kappa\ i \\
\mathcal{K}[\![e_1\ p\ e_2]\!]\ \kappa & = & \mathcal{K}[\![e_1]\!]\ (\lambda x_1.\ \mathcal{K}[\![e_2]\!]\ (\lambda x_2.\ \mathsf{let}\ x_3 = x_1\ p\ x_2\ \mathsf{in}\ \kappa\ x_3)) \\
\mathcal{K}[\![\mathsf{if0}\ e_1\ e_2\ e_3]\!]\ \kappa & = & \mathcal{K}[\![e_1]\!]\ (\lambda x.\ \mathsf{if0}\ x\ (\mathcal{K}[\![e_2]\!]\ \kappa)\ (\mathcal{K}[\![e_3]\!]\ \kappa)) \\
\\
\mathcal{K}_{\text{prog}}[\![e]\!] & = & \mathcal{K}[\![e]\!]\ \ (\lambda x.\ \mathsf{halt}\ x)
\end{array}
$$

Figure 4: CPS conversion

$$
\frac{\Gamma(x) = \tau}{\Gamma \vdash_K x : \tau} \qquad \frac{\Gamma, x{:}\tau \vdash_K e}{\Gamma \vdash_K \lambda x{:}\tau.\, e : \tau \to 0} \qquad \frac{}{\Gamma \vdash_K i : \mathsf{int}} \qquad \frac{\forall i\ .\ \Gamma \vdash_K v_i : \tau_i}{\Gamma \vdash_K (v_1, v_2) : \tau_1 \times \tau_2}
$$

$$
\frac{\Gamma \vdash_K v_1 : \tau \to 0 \quad \Gamma \vdash_K v_2 : \tau}{\Gamma \vdash_K v_1\ v_2} \qquad \frac{\Gamma \vdash_K v : \tau \quad \Gamma, x{:}\tau \vdash_K e}{\Gamma \vdash_K \mathsf{let}\ x = v\ \mathsf{in}\ e}
$$

$$
\frac{\Gamma \vdash_K v : \tau_1 \times \tau_2 \quad \Gamma, x{:}\tau_i \vdash_K e}{\Gamma \vdash_K \mathsf{let}\ x = \pi_i\ v\ \mathsf{in}\ e} \qquad \frac{\Gamma \vdash_K v_1 : \mathsf{int} \quad \Gamma \vdash_K v_2 : \mathsf{int} \quad \Gamma, x{:}\mathsf{int} \vdash_K e}{\Gamma \vdash_K \mathsf{let}\ x = v_1\ p\ v_2\ \mathsf{in}\ e}
$$

$$
\frac{\Gamma \vdash_K v : \mathsf{int} \quad \Gamma \vdash_K e_1 \quad \Gamma \vdash_K e_2}{\Gamma \vdash_K \mathsf{if0}\ v\ e_1\ e_2} \qquad \frac{\Gamma \vdash_K v : \tau}{\Gamma \vdash_K \mathsf{halt}\ v}
$$

Figure 5: Typing rules for values and expressions of λ_K

3.2 Type preservation

The static semantics shown in Fig. 5 defines two typing judgments: $\Gamma \vdash_K v : \tau$ assigns type τ to value v; while $\Gamma \vdash_K e$ asserts that expression e is well typed.

In its simplest form, type preservation states that if a program is well-typed in $\lambda_{\to}$, then the program after CPS conversion will also be well-typed:

Theorem 3.1 *(CPS type preservation) If* $\bullet \vdash e : \tau$*, then* $\bullet \vdash_K \mathcal{K}_{prog}[\![e]\!]$.

In order to prove the above theorem, it is useful to prove a stronger property that establishes the correspondence between the types in $\lambda_{\to}$ and those in λ_K. We can state this correspondence formally as follows:

Theorem 3.2 *($\lambda_{\to}$-λ_K type correspondence) If* $\bullet \vdash e : \tau$*, then* $\bullet \vdash_K \lambda c.\ \mathcal{K}[\![e]\!]\ c : (\mathcal{K}_{type}[\![\tau]\!] \to 0) \to 0$.

Note that the expression in CPS is "wrapped" into a λ-abstraction and thus turned into a value, so that it can be given a type.

3.3 Program representation

Figure 6 shows the typed representation of λ_K. Ideally, we would like to define two mutually recursive types, `ValK` and `ExpK`, representing the syntactic categories of values and expressions, respectively. However, our

```
data V t
data ExpKF a where
    -- values
    KVnum     :: Int                          -> ExpKF (a (V Int))
    KVlam     :: (a (V s) -> a Z)             -> ExpKF (a (V (s -> Z)))
    KVpair    :: a (V s) -> a (V t)           -> ExpKF (a (V (s, t)))
    -- expressions
    Klet_val  :: a (V t) -> (a (V t) -> a Z)         -> ExpKF (a Z)
    Klet_fst  :: a (V (t1, t2)) -> (a (V t1) -> a Z) -> ExpKF (a Z)
    Klet_snd  :: a (V (t1, t2)) -> (a (V t2) -> a Z) -> ExpKF (a Z)
    Klet_prim :: PrimOp -> a (V Int) -> a (V Int) -> (a (V Int) -> a Z)
                                                     -> ExpKF (a Z)
    Kapp      :: a (V (s -> Z)) -> a (V s)           -> ExpKF (a Z)
    KifO      :: a (V Int) -> a Z -> a Z             -> ExpKF (a Z)
    Khalt     :: a (V Int)                           -> ExpKF (a Z)

type ValK a t = Rec ExpKF a (V t)
type ExpK a   = Rec ExpKF a Z
```

Figure 6: Typeful representation of λ_K

fixed point operator (`Rec`, see Fig. 2) can only be applied to a single type, so instead we use the same type for the two syntactic categories. (Alternatively, one might prefer to extend the recursion scheme to the case of two or more types, but we do not attempt this here.)

The distinction between expressions and values is actually not lost: we take advantage of the GADTs to recover this distinction by encoding the corresponding syntactic constraints as type constraints: values have source type `V t` whereas expressions have source type `Z`, so types statically enforce that constructors for values cannot appear where an expression is expected and vice versa.

3.4 Proving type correspondence

At first approximation, by applying the Curry-Howard isomorphism, the type correspondence property of the CPS transform (Theorem 3.2) might be reflected in the type of its implementation in this way:

```
cps :: (∀a. Exp a t) -> (∀a.ValK a 𝒦type⟦t⟧ -> ExpK a) -> (∀a. ExpK a)
```

Here, indeed, we abuse notation by using $\mathcal{K}_{\text{type}}[\![-]\!]$ in a Haskell type expression – we cannot express $\mathcal{K}_{\text{type}}[\![-]\!]$ directly since Haskell lacks intensional type analysis at the level of types. To circumvent the problem, we encode a *proof* of the correspondence between `t` and $\mathcal{K}_{\text{type}}[\![\texttt{t}]\!]$. That is, we instead type `cps` as follows:

```
cps :: (∀a. Exp a t)
       -> ∃cps_t. (CpsForm t cps_t,
                   (∀a. (ValK a cps_t -> ExpK a) -> ExpK a))
```

where a value of type `CpsForm t cps_t` represents a proof that `cps_t` $= \mathcal{K}_{\text{type}}[\![\texttt{t}]\!]$. Such a proof is encoded in a GADT whose data constructors only permit the creation of valid associations between a type in the source language and its corresponding type in CPS form:

```
data CpsForm t cps_t where
  CpsInt  :: CpsForm Int Int
  CpsPair :: CpsForm s cps_s -> CpsForm t cps_t
                -> CpsForm (s, t) (cps_s, cps_t)
  CpsFun  :: CpsForm s cps_s -> CpsForm t cps_t
                -> CpsForm (s -> t) ((cps_s, cps_t -> Z) -> Z)
```

Now, since we use HOAS, we have to structure the CPS transformation slightly differently: we will define a function that performs CPS conversion of a single node, and apply the iterator to this function in order

to obtain a function that converts an entire program (like we did in the pretty-printer example of Sec. 2.2.) The type of the function performing CPS conversion of an individual node has the following type:

```
cpsAux :: ∀a. ExpF (CPS a t) -> CPS a t
```

where `CPS a t` represents the CPS-converted form of an expression of source type `t`, and is an abbreviation whose meaning is defined as follows:

```
type CPS a t =
    ∃cps_t. (CpsForm t cps_t,
             ((ValK a cps_t -> ExpK a) -> ExpK a))
```

To illustrate the technique, the case that CPS-converts a pair construction term (a, b) is implemented as follows:

```
cpsAux (Pair (a::CPS a s) (b::CPS a t)) =
  case (a, b) of
    ((s_cps_s, cps_a), (t_cps_t, cps_b)) ->
      ((CpsPair s_cps_s t_cps_t),
       (λk -> (cps_a (λv1 -> cps_b (λv2 -> k (pairK v1 v2))))))
```

As can be seen from this example, the code follows the structure of an inductive proof, where the CPS transformation and its proof of type-preservation are interlaced.

Finally, the main function of the CPS transformation:

```
cpsProg :: (∀a. Exp a t) -> (∀a. ExpK a)
```

is obtained by applying the iterator to the function `cpsAux`. (Since it implements $\mathcal{K}_{\text{prog}}[\![-]\!]$, its type does not reflect the type correspondence property, only type preservation.)

3.5 Functional dependency

In some places of the type correspondence proof, we need to use the fact that the CPS form of a given type in $\lambda_{\rightarrow}$ is unique, that is:

Theorem 3.3 *(Uniqueness of CPS form) If* $\mathcal{K}_{type}[\![\tau]\!] = \tau_K$ *and* $\mathcal{K}_{type}[\![\tau]\!] = \tau'_K$*, then* $\tau_K = \tau'_K$*.*

We refer to this fact as a *functional dependency* between a type τ and its CPS form $\mathcal{K}_{\text{type}}[\![\tau]\!]$, in the sense of [7]. By the Curry-Howard isomorphism we can encode this theorem as a Haskell function. First, we encode type equality using a GADT:

```
data Equal a b where
    Eq_refl :: Equal a a
```

whose only introduction form accounts for reflexivity. Then Theorem 3.3 is proved as follows:

```
cpsUnique :: CpsForm t cps_t -> CpsForm t cps_t' -> Equal cps_t cps_t'
cpsUnique CpsInt CpsInt = Eq_refl
cpsUnique (CpsFun (s_cps_s::CpsForm s cps_s)
                  (t_cps_t::CpsForm t cps_t))
          (CpsFun (s_cps_s'::CpsForm s cps_s')
                  (t_cps_t'::CpsForm t cps_t'))
  = case cpsUnique s_cps_s s_cps_s' of
      (Eq_refl::Equal cps_s cps_s') ->
        case cpsUnique t_cps_t t_cps_t' of
          (Eq_refl::Equal cps_t cps_t') ->
             Eq_refl
```

We make use of this theorem, for instance, in the case of function application where we need to use the fact that the CPS form of the argument (`e2`) matches the type expected by the CPS-converted function (`e1`):

```
cpsAux (App (e1::CPS a (s->t))
            (e2::CPS a s)) =
case e1 of
  (CpsFun s_cps_s t_cps_t, m1) ->
    case e2 of (s_cps_s', m2) ->
      case cpsUnique s_cps_s s_cps_s' of Eq_refl ->
        (t_cps_t,
         (λk -> m1 (λr1 -> m2 (λr2 -> appK r1 (pairK r2 (contK k))))))
```

4 Fine points

We discuss here some differences between the previous section and the code we actually use; the problem of unsoundness of our proofs and a way we tried to solve it; as well as how we solve the problem of constructing the HOAS terms, which we have for now conveniently skipped.

4.1 The CPS conversion of Danvy and Filinski

Danvy and Filinski's one-pass CPS conversion [4], where administrative redexes are reduced on-the-fly, can be conveniently expressed using an iterator over a HOAS, as was illustrated in Washburn and Weirich's paper [24]. The essential difference with the conversion shown above is reflected in the representation of a CPS-converted term which, in our setting, would be as follows:

```
type CPS a t =
  ∃cps_t. (CpsForm t cps_t,
           ((ValK a cps_t -> ExpK a) -> ExpK a),     cps-meta
           ((ValK a (cps_t -> Z)) -> ExpK a))        cps-obj
```

A term in CPS is now represented by both (1) a term *cps-meta* parameterized by a meta-level continuation, as before, and (2) a term *cps-obj* parameterized by an object-level continuation, that is, a value of source type `(cps_t -> Z)`. Thus the CPS conversion of a term simultaneously defines these two forms.

We have treated type preservation in the case of the basic CPS transformation in order to simplify the presentation; our compiler actually implements Danvy and Filinski's CPS conversion. The type preservation proof extends to this case without particular difficulty.

4.2 (Un)soundness

One concern with our approach is that the type-preservation proof is encoded in an unsound logic. That is, one can trivially encode a "proof" of type correspondence between any two types `s` and `t` (that is, a value of type `CpsForm s t`) as a non-terminating Haskell term.

At any rate, the compiler could be made to traverse the type-preservation proof after the fact to verify that it is indeed complete – this pass would simply diverge in the event of an incorrect proof.

Of course, one must be careful not to introduce non-terminating terms when developing a proof. The risk is slight, however, the presence of such terms being fairly manifest, and given the fact that *we* are writing the proof. That is, we are not in a PCC setting where the possibility of a malicious adversary exploiting any loop-holes of our logic is a prime consideration. Here, the construction of witnesses is merely a device to verify our intuition. For that purpose, we believe that the degree of confidence provided by our technique is reasonable, although we clearly hope to find something better.

4.3 Haskell type classes

Before resorting to manipulating explicit proofs in an unsound logic, we tried another approach that relied on multi-parameter type classes. This approach initially seemed much more promising and elegant.

The intended use of type classes in Haskell is to control *ad-hoc* polymorphism. A type class can be seen as a predicate asserting the existence of a set of functions defined over that type, the implementation of

these functions being provided as part of an *instance* declaration. For example, the `Show` class states the existence of a `show` function of type `t -> String`, defined for each type `t` that is a member of the class. In the Haskell 98 standard, a type class may involve only a single type argument. However, a common extension supported by Haskell compilers permits the definition of multi-parameter type classes, which extend the notion of predicates over types to that of *relations* among types. Thus, one can declare a type class that represents a relation between types in $\lambda_{\rightarrow}$ and type in λ_K as follows:

```
class CpsForm t cps_t
```

The relation is defined as a set of instance declarations as follows:

```
instance CpsForm Int Int
instance (CpsForm s cps_s, CpsForm t cps_t)
   => CpsForm (s, t) (cps_s, cps_t)
instance (CpsForm s cps_s, CpsForm t cps_t)
   => CpsForm (s -> t) ((cps_s, cps_t -> Z) -> Z)
```

This set of instance of declarations can be viewed as (static) type-level logic programming. Each instance declaration can be read as an inference rule: the first rule is an axiom that states the CPS form of `Int` is `Int`, the second rule states that the CPS form of `(s, t)` is `(cps_s, cps_t)`, provided `cps_s` is the CPS form of `s` and `cps_t` is that of `t`, and similarly for the third rule. Finally, we can express the fact that the relation is a function with an additional clause (a *functional dependency*) to the class declaration as follows:

```
class CpsForm t cps_t | t -> cps_t
```

Now, making use of the type class, we can express type preservation as follows. We'd keep the type of `cpsAux` as before, that is:

```
cpsAux :: ExpF (CPS a t) -> CPS a t
```

but the type synonym `CPS a t` would now stand for the following (existential) type:

```
type CPS a t =
    ∃cps_t. CpsForm t cps_t =>
              (ValK a cps_t -> ExpK a) -> ExpK a
```

Unfortunately, in practice, this scheme doesn't take us very far. GHC isn't currently able to type-check this code, even though it appears logically correct. For this to work, we'd expect the type checker to apply the functinnal dependency and instance declarations to identify the unique type `cps_t` given `t` and to use this information as input for GADT type refinement. But such precise interaction between functional dependencies and GADTs isn't currently present in GHC. The situation may change in the future, if for instance a new internal representation is adopted in GHC [22].

It is worth noting that associated types [1] may provide an attractive alternative to functional dependencies. But in the absence of a robust implementation of associated types, it is unclear at the moment whether we would face the same difficulties as with type classes.

4.4 Construction of higher-order terms

The compiler front-end performs a lexical and syntactic analysis and produces an abstract syntax tree. Here, the abstract syntax tree is a term in higher-order abstract syntax. Constructing an efficient representation of such higher-order terms is the subject of some concern. To illustrate, suppose that one attempts to construct a parser that directly produces a higher-order representation; then one invariably ends up writing a parser having essentially this form:

```
parse ... = case ... of
               ... -> Lam (λx -> ... (parse x ...) ... )
               ...
```

The problem is that the body of the function may indeed refer to the newly bound variable (`x`), so the variable has to be passed as argument to `parse` in the recursive call. Thus the resulting syntax tree contains a call to `parse` under every `Lam` node, with dramatic consequences on the compiler's performance.

Fortunately, there is a simple solution to this problem. A higher-order representation can be constructed by *meta-programming*, that is, by using an extension of Haskell through which fragments of Haskell code can be manipulated under program control. We make use of Template Haskell [20], a meta-programming facility now included in GHC.

In our compiler, we use a parser producing a first-order abstract syntax, and then turn it into a HOAS term using Template Haskell. The first-order syntax trees are represented in a conventional manner:

```
data AST where
  Fvar :: Ident -> AST
  Flam :: Ident -> AST -> AST
  Fapp :: AST -> AST -> AST
  ...
```

where `Ident` is a type for identifiers. We define a Template Haskell function `lift` that turns this representation into HOAS:

```
lift :: AST -> ExpQ
lift (Fvar x)     = varE (mkName x)
lift (Flam x t b) = [| Lam $(lam1E (varP (mkName x)) (lift b)) |]
lift (Fapp a b)   = [| App $(lift a) $(lift b) |]
...
```

The type `ExpQ` is a type defined by Template Haskell for representing Haskell expressions. The code in semantic brackets (`[|−|]`) represents a quoted expression, and the form `$(−)` is used to escape from the quotes (much in the manner of Scheme's `quasiquote` and `unquote`.)

Now, we can apply the above function with the special form `$(lift ast)`. Thus, the main function of the compiler follows this structure:

```
compile :: ProgramText -> Assembly
compile program_text =
   let ast = parse program_text
       exp = $(lift ast)
   in (generate_code . closure_conversion . cps_conversion) exp
```

In essence, `lift` rewrites the source program in Haskell, in terms of the constructors that define our HOAS representation. If the resutling Haskell code is well-typed, then so is the source program – thus we also get a source-level type-checker for free, courtesy of GHC.

5 Related work

There has been a lot of work on typed intermediate languages, beginning with the TIL [23] and FLINT [17, 16] work, originally motivated by the optimizations opportunities offered by the extra type information. [12] introduced the idea of Proof-Carrying Code, making it desirable to propagate type information even further than the early optimization stages, as done in in [11].

In [19], Shao et al. show a low-level typed intermediate languages for use in the later stages of a compiler, and more importantly for us, they show how to write a CPS translation whose type-preservation property is statically and mechanically verified, like ours.

In [13], Emir Pasalic develops a statically verified type-safe interpreter with staging for a language with binding structures that include pattern matching. The representation he uses is based on deBruijn indices and relies on type equality proofs in Haskell.

In [2], Chiyan Chen et al. also show a CPS transformation where the type preservation property is encoded in the meta language's type system. They use GADTs in similar ways, including to explicitly manipulate proofs, but they have made other design tradeoffs: their term representation is first order using

deBruijn indices, and their implementation language is more experimental. In a similar vein, Linger and Sheard [10] show a CPS transform over a GADT-based representation with deBruijn indices; but in contrast to Chen's work and ours, they avoid explicit manipulation of proof terms by expressing type preservation using type-level functions.

In [9], Leroy shows a backend of a compiler written in the Coq proof assistant, and whose correctness proof is completely formalized. He uses a language whose type systems is much more powerful than ours, but whose computational language is more restrictive.

In [5], Fegaras and Sheard show how to handle higher-order abstract syntax, and in [24], Washburn and Weirich show how to use this technique in a language such as Haskell. We use this latter technique and extend it to GADTs and to monadic catamorphisms.

GADTs were introduced many times under many different names [25, 3, 21]. Their interaction with type classes is a known problem in GHC and a possible solution was proposed in [22].

6 Discussion and future work

The use of HOAS raises concerns about the performance of the compiler. There is a question whether it will incur a significant amount of repeated work, as would have been the case in the parser had we not used Template Haskell. The answer wholly depends on the structure of the compiler: if it is streamlined to the point that each intermediate representation is used only once, then performance won't suffer much. But repeated analysis phases over the same intermediate representation would clearly result in repeated work. In this case, we'd simply use Template Haskell again to "flatten" the representation after certain phases and thus recover viable performance.

Of course we intend to add many more compilation phases, such as closure conversion, optimization, register allocation, to make it a more realistic compiler. Closure conversion in particular offers a greater challenge than CPS since it is somewhat more intensive w.r.t. program analysis. The type of a code fragment (at least locally, i.e. within a closure) depends on its free variables. This mean some program analysis will have to take place statically in order to be reflected in Haskell's type system.

We also intend to make our source language more powerful by adding features such as parametric polymorphism and recursive types.

Also we hope to find some clean way to move the unsound term-level manipulation of proofs to the sound type-level.

In the longer run, we may want to investigate how to generate PCC-style proofs. Since the types are not really propagated any more during compilation, constructing a PCC-style proof would probably need to use a technique reminiscent of [6]: build them separately by combining the source-level proof of type-correctness with the verified proof of type preservation somehow extracted from the compiler's source code.

6.1 Conclusion

We have shown how to write some parts of a compiler using GADTs such that the type system of the language in which the compiler is written can automatically verify that the compiler properly preserves the types of its programs. We have specifically shown how to write the CPS conversion and the conversion from an untyped representation to a typed representation.

As part of this, we have shown how to integrate generalized algebraic data types with Washburn and Weirich's technique to encode higher-order abstract syntax in a Haskell-like language. We have also shown how to use Template Haskell to leverage Haskell's type checker to do our type checking for us.

References

[1] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–13, New York, NY, USA, 2005. ACM Press.

[2] Chiyan Chen and Hongwei Xi. Implementing typeful program transformations. In *PEPM '03: Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 20–28, New York, NY, USA, 2003. ACM Press.

[3] James Cheney and Ralf Hinze. First-class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, 2003.

[4] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[5] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Conf. Record 23rd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages, POPL'96, St. Petersburg Beach, FL, USA, 21–24 Jan. 1996*, pages 284–294. ACM Press, New York, 1996.

[6] Nadeem Abdul Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. In *Annual Symposium on Logic in Computer Science*, pages 89–100, Copenhagen, Denmark, July 2002.

[7] Mark P. Jones. Type classes with functional dependencies. *Lecture Notes in Computer Science*, 1782:230–244, 2000.

[8] Xavier Leroy. Unboxed objects and polymorphic typing. In *Symposium on Principles of Programming Languages*, pages 177–188, January 1992.

[9] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Symposium on Principles of Programming Languages*, pages 42–54, New York, NY, USA, January 2006. ACM Press.

[10] Nathan Linger and Tim Sheard. Programming with static invariants in omega. Unpublished, 2004.

[11] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[12] George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, jan 1997.

[13] Emir Pasalic. *The Role of Type Equality in Meta-Programming*. PhD thesis, Oregon Health and Sciences University, The OGI School of Science and Engineering, 2004.

[14] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 199–208, New York, NY, USA, 1988. ACM Press.

[15] Zhong Shao. Flexible representation analysis. In *International Conference on Functional Programming*, pages 85–98. ACM Press, June 1997.

[16] Zhong Shao. An overview of the FLINT/ML compiler. In *International Workshop on Types in Compilation*, June 1997.

[17] Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Symposium on Programming Languages Design and Implementation*, pages 116–129, La Jolla, CA, June 1995. ACM Press.

[18] Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *International Conference on Functional Programming*, pages 313–323. ACM Press, September 1998.

[19] Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. A type system for certified binaries. In *Symposium on Principles of Programming Languages*, pages 217–232, January 2002.

[20] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16, New York, NY, USA, 2002. ACM Press.

[21] Tim Sheard and Emir Pasalic. Meta-programming with built-in type equality. In *Logical Frameworks and Meta-Languages*, Cork, July 2004.

[22] Martin Sulzmann, Manuel M. T. Chakravarty, and Simon Peyton Jones. System F with type equality coercions. Submitted to ICFP'06.

[23] David Tarditi, Greg Morrisett, Perry Cheng, Christopher Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *Symposium on Programming Languages Design and Implementation*, pages 181–192, Philadelphia, PA, May 1996. ACM Press.

[24] Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 249–262, Uppsala, Sweden, August 2003. ACM SIGPLAN.

[25] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Symposium on Principles of Programming Languages*, pages 224–235, New Orleans, LA, January 2003.

Type-level Computation Using Narrowing in Ωmega.

Tim Sheard
Computer Science Department
Maseeh College of Engineering and Computer Science
Portland State University

1 Introduction

There has been a lot of recent interest in exploiting the Curry-Howard isomorphism in type systems for more or less traditional programming languages. Types based upon the Curry-Howard isomorphism can express precise properties of programs. Such properties can be either functional or non-functional properties. I.e. they can constrain either the output of the program, or the resources needed to produce the output, or both.

The Curry-Howard isomorphism states that types are propositions (or properties), and that programs are proofs. This isomorphism points out two views of the same phenomena. When we write $prog_1 :: type_1$, we are stating both that $prog_1$ has type $type_1$, and $prog_1$ is a proof of the property $type_1$. By generalizing the notion of type, we can use types to be precise about how programs behave.

To use this concept effectively, we have to think of types as more than just a mechanism to predict the layout or representation of data, or as a mechanism to classify values into similar groups (all the integers, all the booleans, etc.). In Ωmega, the normal types such as `Int`, `String`, and the function arrow (`->`), play these roles, but programmers can also introduce new kinds of types, which look like algebraic data structures (list, trees, ect), but which live at the type level, rather than at the value level like ordinary algebraic data. Such types play new roles, not played by traditional types. They do not classify values, but instead they are used to describe properties such as the shape of value-level data, or the resources consumed by a function application. In addition, Ωmega allows programmers to write programs that manipulate these types, and these type-manipulating programs are executed, not at run-time, but instead at type-checking time to direct the type-checking process. Declaring a type for a program states a property of that program, and checking that a program has a declared type, proves it has that property.

Our goal is to build a system in which the specification of designs, the definition of properties, the implementation of programs, and the checking that programs adhere to their properties, are all bundled in a coherent manner into a single unified system that appears to the user to be a programming language. We hope to use Ωmega as broad spectrum language, capable of handling abstract properties as well as implementation minutiae, where the connection between properties and programs is formal and precise.

The language Ωmega is our first attempt at building such a system. It is a work in progress [23, 24, 29, 33, 32, 30, 31, 34]. While it does not yet meet all of the goals we will list below, we have made it freely available (`http://www.cs.pdx.edu/~sheard/Omega/index.html`).

2 Related work

Writing programs at both the value and type level is a strange but exciting idea indeed! The programmer writes programs at two levels, one to execute at run-time, the other to be executed by the type checker at compile-time to enforce properties. The design space for such a system is large. Explored corners of the design space include logical systems such as Inductive Families[12, 15], theorem provers (Coq[39], Isabelle[25]), logical frameworks (Twelf[27], LEGO[20]), and proof assistants (ALF[22], Agda[11]).

Recently, the design space exploration has grown to include work that uses dependent types to build "practical" systems that are part language, part reasoning system. The designs of these systems vary widely. Some explore different parts of the design space, and most have some explicit design goals that the language designers were trying to accomplish. We first list our own design goals for Ωmega, and then try and place some of the other systems within the design space. In Ωmega, our design goals include:

- We want both programmers and logicians to be able to use the system. We want our system to both look and behave like a programming language, but still be a sound logic.
- When the type system disallows a program we want the programmer to be able to understand why the program was rejected, and to be able to understand how to repair the program if possible. (This is very hard!)
- There should be a strict separation between values and types. This allows the implementation to use a type erasure semantics, so that types (properties) cost nothing at run-time. This is an important instance of a more general policy . . .
- There should be a pay for what you use approach. Programs which need not be too precise, and hence use little or no logic, should incur little or no cost. Ones whose properties must be highly constrained, will be more costly to write and maintain.
- The language must be capable of expressing all kinds of computations. We should not throw away language features because they might make the type system unsound. Instead we should use the type system to *separate* sound features from unsound ones. The type system should clearly mark the boundaries between the two. It should spell out the obligations required to control the unsound features, and support and track how these obligations can be met.

A partial list of contemporary approaches to combining programming languages and logics include:

- **Cayenne.** Augustsson's Cayenne language[5, 4] is a dependently typed version of Haskell. Like many dependently typed systems it does not distinguish between values and types.
- **Epigram.** McBride's Epigram[21] system is a contemporary attempt at combining practical programming language with reasoning system by using dependant types with out a strict phase separation. The system is an integrated editor type-checker. The feel of using Epigram is more akin to using a proof editor, than writing a program.
- **RSP.** Stump's Rogue-Sigma-Pi[37, 40] combines dependent types, general recursion, and imperative features in a type safe way. The emphasis is on an imperative, rather than a pure functional approach is emphasized.
- **DML.** Xi and Pfenning's Dependent ML[43, 14] is an ML like language that allows the user to define type refinements. DML, like Ωmega separates types from values, but the set of indexed types cannot be extended by the programmer, and constraints are solved by a fixed decision procedure (rather than be user authored type level functions).
- **ATS.** Chen and Xi have developed the a paradigm they call Applied Type Systems[42, 10, 9]. It shares many of Ωmega's design goals, and is very closely related.
- **Constraint Handling Rules.** Martin Sulzman has applied his general mechanism for defining constrained type systems, Constraint Handling Rules, to build a system with similar goals[38, 36]. His system has some nice completeness results which we need to learn more about.

The technical focus of this paper is to describe a previously unexplored mechanism for handling functions at the type level: narrowing. The non-technical focus is to introduce several non-trivial examples which illustrate how types can describe properties. These examples were also chosen because they work well with the narrowing approach. We will also discuss several problems for which the narrowing approach is not effective, and to suggest some future work.

3 Combining Programming Language and Logic in Ωmega

We have adopted the following structure for the Ωmega language. Ωmega is a language with an infinite hierarchy of computational levels: value, type, kind, sort, etc. Computation at the value level is performed by reduction, and is largely unconstrained. Computation at all higher levels is performed by narrowing, and is constrained in several ways. First, all "data" at the type level and above is inductively defined data (no floats or primitive data for example). Second, functions at the type level and above must be inductively sequential. This is a constraint on the form of the definition, not on the expressiveness of the language. See Section 8 for a detailed discussion of narrowing and these constraints.

Terms at each level are classified by terms at the next level. Thus values are classified by types, types are classified by kinds, kinds are classified by sorts, etc. We maintain a strict phase distinction – the classification of a term at level n cannot depend upon terms at lower levels. For example, no types can depend on values, and no kinds can depend on types. Programmers are allowed to introduce new terms and functions at every level, but any particular program will have terms at only a finite number of levels.

We formalize properties of programs by exploiting the Curry-Howard isomorphism. Terms at computational level n, are used as proofs about terms at level $n + 1$. We use indexed types to maintain a strict and formal connection between the two levels, and singleton types to maintain the strict separation between values and types (more about this in Section 5).

4 A simple example

To illustrate the hierarchy of computational levels we give the following two-level example. We introduce tree-like data (the natural numbers, `Nat`) at the type level by using the `kind` introduction form. We write a function at the type level over this data (`plus`). At the type level and higher, we distinguish function application from Constructor application by surrounding function application by braces (`{` and `}`).

```
kind Nat:: *1 where
  Z:: Nat
  S:: Nat ~> Nat

plus:: Nat ~> Nat ~> Nat
{plus Z m} = m
{plus (S n) m} = S {plus n m}
```

We then introduce data at the value level (`Seq`) using the `data` introduction form. The types of such values are indexed by the natural numbers. These indexes describe an invariant about the constructed values – their length appears in their type – consider the type of `l1`. Finally, we introduce a function at the value level over `Seq` values (`app`). The type of `app` describes one of its important properties – there is a functional relationship between the lengths of its two inputs, and the length of its output.

```
data Seq:: *0 ~> Nat ~> *0 where
  Snil :: Seq a Z
  Scons:: a -> Seq a n -> Seq a (S n)

l1 = Scons 3 (Scons 5 Snil) :: Seq Int (S(S Z))

app:: Seq a n -> Seq a m -> Seq a {plus n m}
app Snil ys = ys
app (Scons x xs) ys = Scons x (app xs ys)
```

Natural numbers at the type level have proved to be very useful. So useful, in fact, that we have added special syntactic sugar for constructing them. We sometimes write `#0` for `Z`, and `#1` for `(S z)`, and `#2` for `(S (S Z))`, etc. We may also write `#(1 + n)` for `(S n)`, and `#(2 + n)` for `(S (S n))`, etc. when `n` is a variable.

5 Introduction to Ωmega

Throughout this section we introduce the features of Ωmega by comparing and contrasting them with the features of Haskell. We assume a basic understanding of Haskell programs.

Feature: Kinds. Kinds introduce new tree-like data at the type level and higher. A kind declaration introduces both the constructors for the tree-like data and the object that classifies these structures. Consider the two new kinds `Set` and `Termination`:

```
kind Set:: *1 where
  Univ:: Set
  Deny:: Set
  Times:: Set ~> Set ~> Set
  Plus:: Set ~> Set ~> Set

kind Termination:: *1 where
  Total:: Termination
  Partial:: Termination
```

The new tree-like data at the type level are constructed by the type-constants (`Univ`, `Total`, and `Partial`), and type constructors (`Times`, and `Plus`). The kinds `Set` and `Termination` classify these structures, as shown explicitly in the declaration. For example `Univ` is classified by `Set`, and `Plus` is a constructor from `Set` to `Set` to `Set`. Think of the operator `~>` as an function arrow at the type level. Note that while `Univ`, `Total`, or `Partial` live at the type level, there are no values classified by them.

Instead, we use the structures as indexes to value level data, i.e. types like `(T Total)` and `(S Univ Int)`. These indexes will indicate static (type-checking time) properties of values. For example, a value with type `(T Total)` is statically guaranteed to have the `Total` property, and a value of type `(S Univ Int)` is statically guaranteed to have the `Univ` property. We will soon introduce two examples which follow this pattern.

The first example is a data structure representing expressions of some object language, where an Ωmega term of type `(Exp Total)` represents an object level term that always terminates when executed, and a term of type `(Exp Partial)` represents an object level term which may not terminate. Note that the index is a semantic invariant that appears in the type of every `Exp`. Judicious use of indexes in the definition of `Exp` can ensure that every well-typed term maintains the invariant.

The second example classifies values representing patterns by types like `(Pattern Univ)` or `(Pattern Deny)`. Semantically, a pattern is a predicate over some domain. I.e. some subset of that domain. For example, in Haskell we use patterns in case expressions. The pattern `(5,x)` describes elements of the domain of pairs of integers such as `(5,2)` and `(5,9)` but *not* `(3,5)`.

We will use the `Set` index as a static invariant describing what sets in the domain are matched by a pattern. Consider a domain of simple objects, products, and sums. An index of `Univ` means the pattern matches every value. An index of `Deny` means their exists at least one value the pattern does not match, perhaps many. Indexes of `(Times a b)` means the pattern matches a product, and the left part of the product matches `a`, and the right part of the product matches `b`. Similar reasoning holds for disjoint sums with the `(Plus a b)` index.

Feature: GADTs. Generalized Algebraic Datatypes allow constructor functions to have more general types than the types supported by the `data` declaration in Haskell. GADTs are important because the additional generality allows programmer to express properties of types using type indexes and witnesses (or proof) objects. GADTs are the machinery that support the Curry-Howard isomorphism in Ωmega.

The `data` declaration in Ωmega defines generalized algebraic datatypes (GADT), a generalization of the algebraic datatypes available in Haskell. This is characterized by explicitly classifying constructors in a `data` declaration with their full types. The additional generality arises because the range of a constructor in a GADT is not constrained to be the type constructor applied to only type variables. For example consider the `Pattern` value level type:

```
data Pattern:: Set ~> *0 ~> *0 where
  PVar :: String -> Pattern Univ t
  PInt :: Int -> Pattern Deny Int
  PPair:: Pattern s1 a ->
          Pattern s2 b ->
          Pattern (Times s1 s2) (a,b)
  PWild:: Pattern Univ t
  PNil :: Pattern (Plus Univ Deny) [t]
  PCons:: Pattern s1 a ->
          Pattern s2 [a] ->
          Pattern (Plus Deny (Times s1 s2)) [a]
```

Note, that instead of ranges like `(Pattern a b)`, where only type variables like `a` and `b` can be used as parameters, the ranges contain sophisticated instantiations such as `(Pattern Univ t)`. Note that the first index to `Pattern` (the one of kind `Set`) is used to describe an invariant about the meaning of a pattern. A variable pattern `(PVar "x")::  Pattern Univ t` matches all values of all types t. But, consider the nested pattern:

```
(PCons (PInt 4) (PVar "x")):: Pattern (Plus Deny (Times Deny Univ)) [Int]
```

Its type tells us a a lot. It should be matched against terms of type `[Int]`. Such objects can be considered disjoint sums. The pattern definitely does not match a left injection (that is reserved for `Nil` objects). It matches some right injections (i.e. a `Cons` object) but not all. A `Cons` object can be considered a product. The right part of the product (the head of the list) only matches some things (i.e. integers with value 4, but not other integers such as 3 or 7). And the right part of the product (the tail) matches everything. All this is embedded in the type of the term! The GADT `Pattern` as an abstraction of the more complex GADT `Pat` we will see in Section 10.

Feature: Type functions. Kind declarations allow us to introduce new tree-like structures at the type level. We can use these structures to parameterize data at the value level as we did with `Pattern`, or we can compute over these tree like structures. Such functions are written by pattern matching equations, in much the same manner one writes functions over data at the value level. Several useful functions over types classified by `Termination` and `Set` are:

```
lub:: Termination ~> Termination ~> Termination
{lub Total x} = x
{lub Partial x} = Partial

allU:: Set ~> Termination
{allU Univ} = Total
{allU Deny} = Partial
{allU (Times x y)} = {lub {allU x} {allU y}}
{allU (Plus x y)} =  {lub {allU x} {allU y}}

union:: Set ~> Set ~> Set
{union Univ a} = Univ
{union (Times p1 p2) (Times q1 q2)} = Times {union p1 q1} {union p2 q2}
{union (Times a b) Deny} = Times a b
{union (Times a b) Univ} = Univ
{union (Plus p1 p2) (Plus q1 q2)} = Plus {union p1 q1} {union p2 q2}
{union (Plus p1 p2) Deny} = Plus p1 p2
{union (Plus p1 p2) Univ} = Univ
{union Deny x} = x
```

Like functions at the value level, the type functions `lub`, `allU` and `union` are expressed using equations. At the type level, we use brackets (`{}`) to surround function application to distinguish it clearly from

constructor application at the type level (like `(Tree Int)` or `(Pattern Univ Int)`). The function `lub` is a binary function that combines two `Terminations`, and `allU` converts a `Set` into a `Termination`. It returns `Total` only if all parts of the `Set` are universal. Both functions are strict total (terminating) functions. The type function `union` unions two `Set`s.

Pattern: Singleton Types. Sometimes it is useful to direct computation at the type level, by writing functions at the value level. Even though types cannot depend on values, this is possible by the use of singleton types. The idea is to build a completely separate isomorphic copy of the type in the value world, but still retain a connection between the two isomorphic structures. This connection is maintained by indexing the value-world type with the corresponding type-world kind. This is best understood by example. Consider reflecting the kind `Nat` into the value-world by defining the type constructor `SNat` using a `data` declaration.

```
data SNat:: Nat ~> *0 where
  Zero:: SNat Z
  Succ:: SNat n -> SNat (S n)

three = (Succ (Succ (Succ Zero))):: SNat(S(S(S Z)))
```

Here, the value constructors of the `data` declaration for `SNat` mirror the type constructors in the `kind` declaration of `Nat`. We maintain the connection between the two isomorphic structures by the use of `SNat`'s natural number index. This type index is in one-to-one correspondence with the shape of the value. Thus, the type index of `SNat` exactly mirrors its shape. For example consider the example `three` above, and pay particular attention to the structure of the type index, and the structure of the value with that type.

We call such related types singleton types because there is only one element of any singleton type. For example only `Succ (Succ Zero)` inhabits the type `SNat(S (S Z))`. It is possible to define a singleton type for any first order type (of any kind). All Singleton types always have kinds of the form `I ~> *0` where `I` is the index we are reflecting into the value world. We sometimes call singleton types *representation types*. We cannot over emphasize the importance of the singleton property. Every singleton type completely characterizes the structure of its single inhabitant, and the structure of a value in a singleton type completely characterizes its type. Thus we can compute over a value of a singleton type, and the computation at the value level can express a property at the type level.

By using singleton types we completely avoid the use of dependent types where types depend on values[35, 28].

Pattern: A pun: Nat'. We now define the type `Nat'`, which is in all ways isomorphic to the type `SNat`. The type `Nat'` is also a singleton type representing the natural numbers, but it relies on an anomaly of the Ωmega type system. In Ωmega (as in Haskell) the name space for values is separate from the name space for types. Thus it is possible to have the same name stand for two things. One in the value space, and the other in the type space. We exploit this ability by defining the pun `Nat'`.

```
data Nat':: Nat ~> *0 where
  Z:: Nat' Z
  S:: Nat' n -> Nat' (S n)

three' = (S(S(S Z))):: Nat'(S(S(S Z)))
```

The value constructors `(Z:: Nat' Z)` and `(S:: Nat' n -> Nat' (S n))` are ordinary values whose types mention the type constructors they pun. In `Nat'`, the singleton relationship between a `Nat'` value and its type is emphasized even more strongly, as witnessed by the example `three'`. Here the shape of the value, and the type index appear isomorphic.

We further exploit this pun, by extending the syntactic sugar for writing natural numbers at the type level (`#0`, `#1`, etc.) to their singleton types at the value level. Thus we may write `(#2:: Nat' #2)`.

Pattern: Computing Programs and Properties Simultaneously. We can write programs that compute an indexed value along with a witness that the value has some additional property. For example,

when we add two static length lists, the resulting list has a length that is related to the lengths of the two input lists, and we can simultaneously produce a witness to this relationship.

```
data Sum:: Nat ~> Nat ~> Nat ~> *0 where
  PlusZ:: Sum Z m m
  PlusS:: Sum n m z -> Sum (S n) m (S z)

app1:: Seq a n -> Seq a m -> exists p . (Seq a p,Sum n m p)
app1 Snil ys = Ex(ys,PlusZ)
app1 (Scons x xs) ys = case (app1 xs ys) of { Ex(zs,p) ->  Ex(Scons x zs,PlusS p) }
```

`Ex` is the "pack" operator of Cardelli and Wegner[8]. It turns a normal type `(Seq a p,Plus n m p)` into an existential type `(exists p.(Seq a p,Plus n m p))`.

Feature: Tags and Labels. Many systems have a notion of label. Usually, labels are unique names that support a notion of equality. Many times it is convenient to think of every label has having unique type in a family of related types. As a *first approximation*, consider the finite kind `Tag` and its singleton type `Label`:

```
kind Tag = A | B | C | D
```

```
data Label:: Tag ~> *0 where
  A:: Label A
  B:: Label B
  C:: Label C
  D:: Label D
```

Here, we again deliberately use the value/type name space overloading we used in the example `Nat'`. The names `A`, `B`, and `C` are defined in both the value and type name spaces. They name different, but related objects in each space. At the value level, every `Label` has a type index that reflects its value. I.e. `A::Label A`, and `B::Label B`, and `C::Label C`. So in the value name space `A` is an ordinary value, and because it is a singleton, its type `(Label A)` reflects its structure. In the type `(Label A)`, the `A` is different (but related) object that lives at the type level, and is classified by the kind `Tag`. This is a very useful pun. Every label at the value level has a different type. Thus we can distinguish statically that two labels differ (or are the same) by observing their type.

The problem with the approach above is that there are only four different labels. We would like a countably infinite set of labels. We can't define this explicitly, but we can build such a type as a primitive inside of Ωmega. At the type level, every legal identifier whose name is preceded by a back-tick (`) is a type classified by the kind `Tag`. For example, at the type level `` `abc `` is classified by `Tag`. At the value level, every such symbol `` `abc `` is also reflected, but here it is classified by the singleton type `(Label `abc)`. We write: `` `abc :: Label `abc :: Tag ``. We say the value `` `abc `` is classified by the singleton type `(Label `abc)` which is classified by the kind `Tag`. Labels are singleton types, and as such their values are uniquely determined by their type. Their values are reflected in their types. This is a powerful mechanism we will exploit in the next section.

Pattern: Binding in object languages. Binding of variables is a perennial problem when modeling rich languages. Our approach to binding is to use a two prong approach, using both labels and de Bruijn indices. Such an approach is related to the notion of record structures. Two define records, we proceed in two steps. We define a new kind at the type level called rows, then we define a new data structure (records) at the value level that is classified by rows.

```
kind Row:: *1 ~> *1 where
  RNil:: Row a
  RCons:: a ~> Row a ~> Row a
kind HasType = Has Tag *0
```

Rows are nothing more than polymorphic list like structures at the type level. Examples of `Row`s include: `(RCons Int (RCons Bool RNil))` a row of types and `(RCons #5 RNil)` a row of `Nat`, and `(RCons (Has `name String))` a row of `HasType`. Such types are kinded by `Row`. The kind `Row` is a higher order kind, and is classified

by `(*1 ~> *1)`. Thus `Row` must be applied to a kind (such as `*0`, `Nat`, or `HasType`) to be well formed (the argument indicates the kind of the types stored in the row).

A `HasType` is nothing more than a pair of types. The the first component of the pair must be a `Tag` and the second component must be an ordinary type (like `Int` or `String`). To define record data at the value level we proceed as follows.

```
data Record:: Row Hastype ~> *0 where
  RecNil :: Record RNil
  RecCons:: Label a -> b -> Record l -> Record (RCons (Has a b) l)

record1 :: Record (RCons (Has `name String) (RCons (Has `age Int) RNil))
record1 = RecCons `name "Tim" (RecCons `age 21 RecNil)
```

Record data is constructed by applying the constructors `RecCons` and `RecNil` to appropriate values. See the example `record1` above. Note the use of labels (`` `name `` and `` `age ``) at the value level, whose kinds are reflected at the type level in the record type.

Labeled de Bruijn indices are implemented by using exactly the same structure. We build a value level data structure `Exp` indexed by `(Row HasType)`. This index uniquely identifies the variables appearing in the term, in the terms type. If the language has binding structure (i.e. any kind of scoping mechanism like lambda or let), then the binding location of a variable is determined by its location in the environment (this is what makes it de Bruijn-like). For example let the value level data `Exp` have two indexes: `Exp:: Row HasType ~> *0 ~> *0`. Interpret a term with type `(Exp r t)` as a term with type `t` whose free variables are described by `r`. For example, consider a term with the following type:

```
Exp (RCons (Has `f (b -> t)) (RCons (Has `x b) d)) t
```

This term has two free variables, named by `` `f `` and `` `x ``, with types `(b -> t)` and `b`, respectively. This term should occur inside a larger context where the variable `` `f `` is the most closely nested variable, and the variable `` `x `` is the the second most closely nested variable. This hybrid approach accomplishes two things. The labels are there for the reader, but the location or de Bruijn index is what counts. This makes alpha-renaming unnecessary when formally manipulating the terms, but the labels provides the reader with some visual clues when inspecting terms and their types. `Tag`s, `Label`s, `Row`s, and `Hastype` build objects that look very much like typing environments.

We have found `Row` and `HasType` so useful we have built special syntactic sugar for printing them. For example, `Rec(RCons (Has `x Int) (RCons (Has `a Bool) RNil))` prints as `Rec {`x:Int,`a:Bool}`. The syntactic sugar for `Row` and `HasType` replaces `RCons` and `RNil` with squiggly brackets, and replaces `Has` with colon. A type classified by `Row` whose (ultimate) tail is not `RNil` (i.e. a type variable) prints with a trailing semi-colon. For example,
`Rec(RCons (Has `x Int) (RCons (Has `a Bool) w))` prints as `Rec {`x:Int,`a:Bool; w}`. Using this notation, our earlier example looks like: `Exp {`f: b->t, `x: b; d} t`

Now that we understand the mechanism, lets use it to define object-languages with binding structures that track their free variables in their meta-level types. The object-language `(Lam env t)` represents the simply typed lambda calculus.

```
data Lam:: Row HasType ~> *0 ~> *0  where
  Var      :: Label s -> Lam (RCons (Has s t) env) t
  Shift    :: Lam env t -> Lam (RCons (Has s q) env) t
  Abstract :: Label a -> Lam (RCons (Has a s) env) t -> Lam env (s -> t)
  Apply    :: Lam env (s -> t) -> Lam env s -> Lam env t
```

The first index to `Lam`, `env` is a Row tracking its variables, and the second index, `t` tracks the object-level type of the term. For example, a term with variables `x` and `y` might have type `Lam {`x:Int, `y:Bool; u} Int`.

The key to this approach is the use of `Row` and `Hastype` in the typing of functions (`Abstract`) and variables (`Var`). Consider the `Var` constructor function. To construct a variable we simply apply `Var` to a label, and its type reflects this. For example, here is the output from a short interactive session with the Ωmega interpreter.

```
prompt> Var ‘name
(Var ‘name)::
forall a (u:Row HasType) . Lam {‘name:a; u} a

prompt> Var ‘age
(Var ‘age)::
forall a (u:Row HasType) . Lam {‘age:a; u} a
```

Variables behave like Bruijn indices. Variables created with `Var` are like the natural number 0. A variable can be lifted to the next natural number by the successor operator `Shift`. To understand why this is useful consider that the two examples have different names in the same index position. The two variables would clash if they were both used in the same lambda term. To shift the position of variable to a different index, we use the `Shift:: Lam u a -> Lam {v:b; u} a` constructor. Rather than counting with natural numbers (as is done with de Bruijn indices) we “count” with rows, recording both its symbolic name and its type. Here is how we could define two variables `x` and `y` for use in the same environment.

```
x :: Lam {‘x:a; u} a
x = Var ‘x

y :: Lam {u:a,‘y:b; v} b
y = (Shift (Var ‘y))
```

The type system now tracks the variables in an expressions.

```
z :: Lam {‘x:a -> b,‘y:a; u} b
z = (Apply (Var ‘x) (Shift (Var ‘y)))
```

We have found many other useful patterns that exploit the features of Ωmega. There are also several features of Ωmega we have not discussed, notably the use of witness types as static constrains and the constrained type system that manages them. But in the rest of this paper, we focus our attention on the uses of narrowing in Ωmega.

6 The structure of type checking

Ωmega uses a combination of type checking and type inference. Type inference, as implemented in Ωmega, can only assign Hindley-Milner types to terms, and only works over algebraic datatypes (as opposed to generalized algebraic datatypes) so type checking is the normal mode of typing in many Ωmega programs.

During type checking, information can flow in two directions. We can compute the type of a term, then check if it is consistent with its declared type, or we can use the declared type to suggest a type for a term, and then traverse the term computing a type consistent with that suggestion.

In some systems every term has exactly one principal type, all other types for that term are “instances” of that principal type. But such systems may be too restrictive. Checking discovers if a term has a particular type, even when it might also have additional types. The type declaration states which of the multiple types should be chosen as the type for that term. Thus every term is assigned a single type (and hence a single meaning), even if it can have more than one type.

When type checking we often ask if two types (b and c) are mutually consistent ($b \approx c$). Consider a “generic” type rule for function application.

$$\frac{\Gamma \vdash f : c \to d \quad \Gamma \vdash x : b \quad b \approx c}{\Gamma \vdash f \;\; x : d}$$

Type checkers perform computation to answer consistency problems. Different languages ask different kinds of questions. Some questions have simple yes/no answers, others search to find particular values meeting some criteria, or reduce a complicated term into a simple normal form. For example, in a monomorphic language like Pascal, mutually consistent means structurally equal. In a polymorphic language like Haskell, mutually consistent means that b and c are unifiable. In a language with subtyping, like Java, mutually consistent means that b is a subtype of c. In Ωmega, consistent means semantically equivalent. Because of the use of type level functions, syntactic equivalence is no longer sufficient. Types are no-longer static free algebras, but dynamic computations. A very simple example of this phenomena is the `app` function from Section 4. Consider type-checking the second clause.

```
app:: Seq a n ->    Seq a m -> Seq a {plus n m}
app   (Scons x xs)  ys      =  Scons x (app xs ys)
```

From the declaration, we know the pattern `(Scons x xs)` has type `(Seq a (S b))` where (`n = (S b)`), so we must show that the right-hand-side has the type: `Seq a {plus (S b) m}`. Computing the type of the right-hand-side we get: `Scons x (app2 xs ys):: Seq a (S{plus b m})`. The types `(Seq a {plus (S b) m})` and `(Seq a (S {plus b m}))` are consistent only if we can find bindings for `b` and `m` that make `{plus (S b) m}` semantically equal to `(S {plus b m})`. If we allow computation at the type level we will need a computational mechanism that can answer equivalence questions such as `(Seq a {plus (S b) m})` ≈ `(Seq a (S {plus b m}))`.

We know of several choices. First, one can use a combination of reduction and unification. Second, one can use entailment. Third, one can use constraint solving, and Fourth, one can use narrowing.

- **Reduction & unification**. To check for consistency, we reduce both terms to normal form and then unify the normal forms. This is the approach taken by Morrisett [13]. Unfortunately, simple consistency checks such as (`{plus b m}` ≈ `m`) cannot be answered since both sides are in normal form, and unifying does not find the correct solution `b` = `Z`.
- **Entailment**. By expressing computation at the type level logically, we can use entailment mechanisms (such as resolution or higher order pattern matching) to solve consistency checks. This is the case in the logical framework Twelf [26]. This choice is inconsistent with our desire for uniformity in the mechanism used to define functions at the value and type levels.
- **Constraint Solving**. General purpose constraint solving systems can be used to decide equivalence. One example, where constraint solvers is applied to type checking is Sulzman's CHR system[38, 36].
- **Narrowing**. Narrowing combines the power of reduction and unification. Narrowing is a primary computational mechanism of the functional logic language Curry [17, 16]. Unlike unification the terms being compared can contain functions, and unlike reduction the terms being simplified can contain variables. Narrowing finds bindings for some free variables in its terms. Narrowing is a special purpose constraint solving system, that meshes quite well with the equivalence questions asked when there are functions at the type level.

One of our design goals is *uniformity*. The beauty of an integrated system is that users need learn only one tool. If it has many different modes, each requiring different skills, then the benefit of integration is lost. So even if we use a different mechanism to execute programs at compile-time and run-time, we want the user interface to the two to be the same. Narrowing is an excellent choice for execution at type checking time. The interface for both run-time and compile-time computation is the writing of recursive equations.

7 How Narrowing works

Narrowing combines the power of reduction and unification. Narrowing finds bindings for some free variables in the term being narrowed, once instantiated, these bindings allow the term to reduce to a normal form.

If a term contains constructors, function symbols, and variables, it often cannot be reduced. Usually because function calls within the term do not match any left-hand side of any of their definitions. The failure to match is caused by either variables or other function calls in positions where the function definitions have only constructor patterns. Consider narrowing `(plus a Z == Z)` (checking whether `{plus a Z}` is equal to `Z`).

This cannot be reduced because `plus` inducts over its first argument with the patterns `Z` and `(S n)`. But in `(plus a Z == Z)`, the first argument position is a variable `a`. Narrowing proceeds by guessing instantiations for the variable `a` – either `{a -> Z}` or `{a -> (S m)}`, and following both paths.

```
plus:: Nat ~> Nat ~> Nat
{plus Z m} = m
{plus (S n) m} = S {plus n m}
```

guess `{a -> Z}`

```
 ({plus Z Z} == Z)
 (Z == Z)
```

Success !

guess `{a -> (S m)}`

```
 ({plus (S m) Z} == Z)
 (S {plus m Z} == Z)
```

Failure.

The returned solutions are the bindings obtained in every successful path. In the example above we get a list of one solution: `[{a -> Z}]`. Some problems have no solutions, some have multiple, and some even have infinite solutions. Consider `{plus x #2}`, we get `#2` when `{x -> #0}`, and `#3` when `{x -> #1}`, and `#4` when `{x -> #2}` etc.

Narrowing works best when we have a problem with many constraints. The constraints prune the search path resulting in few solutions. If we're type checking with narrowing we hope there is exactly one solution. Consider narrowing `({plus x #3} == #5)`, Guessing `{x -> 0}` and `{x -> S z1}` we get the two paths:

```
{ #3 == #5 }
{ #(1+{plus z1 #3}) == #5 }
```

The first path fails, on the second path we take a single reduction step leaving:

```
{ {plus z1 #3} == #4 }
```

Guessing `{z1 -> 0}` and `{z1 -> S z2}` we get the two paths:

```
{ #3 == #4 }
{ #(1+{plus z2 #3}) == #4 }
```

The first path fails, on the second path we again take a single reduction step leaving:

```
{ {plus z2 #3} == #3 }
```

Guessing `{z2 -> 0}` and `{z2 -> S z3}` we get the two paths:

```
{ #3 == #3 }
{ #(1+{plus z3 #3}) == #3 }
```

The first succeeds, and the second eventually fails, leaving us with only one solution `{ x -> #2 }`. We have found narrowing to be a efficient and understandable mechanism for directing computation at the type level.

8 Narrowing Strategies

While narrowing is non-deterministic, it is both sound and complete with an appropriate strategy[3]. All answers found are real answers, and if there exists an answer, good strategies will find it. When a question has an infinite number of answers, a good implementation will produce these answers lazily. In our type-checking context, finding 2 or more answers is a sign that a program being type checked has an ambiguous type and needs to be adjusted. In the rare occurrence that narrowing appears to diverge on a particular question, we can safely put resource bounds on the narrowing process, declaring failure if the resource bounds

are exceeded. The result of such a declaration, is the possibility of declaring a well- typed function ill-typed. In our experience this rarely happens.

We restrict the form of function definitions at the type level to be inductively sequential[1]. This ensures a sound and complete narrowing strategy for answering type- checking time questions. The class of inductively sequential functions is a large one, in fact every Haskell function has an inductively sequential definition. The inductively sequential restriction affects the form of the equations, and not the functions that can be expressed. Informally, a function definition is inductively sequential if all its clauses are non-overlapping. For example the definition of `zip1` is not-inductively sequential, but the equivalent program `zip2` is.

```
zip1 (x:xs) (y:ys) = (x,y): (zip1 xs ys)
zip1 xs ys = []

zip2 (x:xs) (y:ys) = (x,y): (zip2 xs ys)
zip2 (x:xs) []     = []
zip2 []     ys     = []
```

The definition for `zip1` is not inductively sequential, since its two clauses overlap. In general any non-inductively sequential definition can be turned into an inductively sequential definition by duplicating some of its clauses, instantiating variable patterns with constructor based patterns. This will make the new clauses non-overlapping. We do not think this burden is to much of a burden to pay, since it is applied only to functions at the type level, and it supports sound and complete narrowing strategies.

We pay for the generality of narrowing over unification and reduction by a modest increase in overhead. Narrowing uses a general purpose search algorithm rather than a special purpose unification or reduction engine. Narrowing is Turing complete, so we can solve any problem that can be solved by reduction, and many more.

9 Narrowing and type checking

Ωmega uses a combination of type inference and type checking. Type inference is useful when we are writing programs that do not utilize GADTs or type-level computation. I.e. all the programs we used to write in Haskell. When we use either of these new features Ωmega uses a type checking approach. If a function is given a prototype, declaring its type, it is type checked. If it not given a prototype, we attempt to infer a Hindley-Milner type for it. Our type system is based in large part upon the system described in the paper *Simple unification-based type inference for GADTs* by Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. We also borrow ideas from the paper *Practical type inference for arbitrary-rank type*[18] by Simon Peyton Jones and Mark Shields. Type level functions and narrowing are accommodated with the following extensions.

- In both papers, type checking is described as a monadic computation over the structure of program terms. We strengthen the monad so it is an accumulating monad. I.e. a monad in which a computation can emit an equality constraint between a type and a type function. All such emissions are accumulated into a list of equalities as the computation proceeds.
- Equality constraints between types and type functions are emitted during unification when an attempt is made to unify two terms, at least on of which is a function application at the type level. This is the only source of such constraints.
- At generalization points (at `let` and at top level) the constraints are solved by narrowing. The resulting set of bindings is a unifier, and this is composed with the unifier obtained in the normal type checking process.
- If narrowing fails to return a binding, type checking fails. If narrowing returns more than one binding, the type is ambiguous. Further use of type annotations can often be chosen to pick a unique type from amongst the set of possible types.

10 Tracking termination: An example with lots of narrowing

It is important for proof objects in Ωmega to attest to true things. If Ωmega is regarded as a logic, it should be sound. The only inhabitant of witnesses and proofs should be one of the well formed objects built from its constructors. This requires that expressions with witness types never result in non-terminating or error computations. As we stated in the introduction we hope to use the type system to separate error producing programs from total ones. In this section we introduce an example which is both a rich source of narrowing examples and a framework for future extensions to Ωmega that will separate terminating programs from possibly non-terminating ones using types.

Our strategy is to statically compute both a termination behavior and a type for every Ωmega term. For this we use the kinds `Set` and `Termination` introduced earlier. To track termination across function calls, the type constructor arrow is tagged with a termination behavior. Note that the arrow type constructor would have kind `(->):: *0 ~> Termination ~> *0 ~> *0`. In this future version of Ωmega, we will either write `((->) a b c)` or `(a -b-> c)`. For example:

```
add:: Int -Total-> Int -Total-> Int
divide:: Int -Total-> Int -Partial-> Int
```

where `(-Total->)` is a terminating function, and `(-Partial->)` is a possibly divergent function.

In some preliminary work, which we describe below, we have built a rich model in Ωmega of this future type system for Ωmega. It captures most of the language features in Ωmega, including pattern matching, algebraic data, primitive functions, and recursion. It makes heavy use of indexed GADTs, type functions, and narrowing.

Our strategy is to track both the type and termination behavior as an index of a GADT representing programs terms. An expression of type `(Exp t env p)` is interpreted as representing a program with type `t` in an environment with shape `env`, with termination behavior `p`. An expression with type `(Pat c t e1 e2)` is interpreted as representing a pattern with type `t`, which matches the set of values `c` (a `Set`), and maps an environment with shape `e1`, to one with shape `e2`. Non-termination stems from three causes: non-exhaustive pattern matching, the use of the `undefined` term (think of this as a call to an error function), and recursive functions. The specification makes extensive use of functions at the type level and of narrowing. In our Ωmega model we write `(Arr dom t rng)` for `(dom -t-> rng)`.

```
data Pat:: Set ~> *0 ~> Row HasType ~> Row HasType ~> *0 where
  Pvar :: Label a -> Pat Univ t r (RCons (Has a t) r)
  Ppair:: Pat s1 a g1 g2 -> Pat s2 b g2 g3 -> Pat (Times s1 s2) (a,b) g1 g3
  Pwild:: Pat Univ t r r
  Pnil :: Pat (Plus Univ Deny) [a] r r
  Pcons:: Pat s1 a g1 g2 ->
          Pat s2 [a] g2 g3 ->
          Pat (Plus Deny (Times s1 s2)) [a] g1 g3


data Exp:: *0 ~> Row HasType ~> Termination ~> *0 where
  Var::   Label a -> Exp t (RCons (Has a t) r) Total
  Shift:: Exp t g m -> Exp t (RCons (Has a b) g) m
  Const:: Int -> Exp Int g Total
  Pair::  Exp x g m -> Exp y g n -> Exp (x,y) g {lub m n}
  App::   Exp (Arr x m b) g n -> Exp x g p -> Exp b g {lub {lub p n} m}
  Abs::   Cls xx (Arr a i b) g1 -> Exp (Arr a {lub {allU xx} i} b) g1 tt
  Nil::   Exp [a] g Total
  Cons::  Exp a g m -> Exp [a] g n -> Exp [a] g {lub m n}
  Error:: Exp a g Partial
  Let::   Label f ->
          Exp (Arr a Partial b) (RCons (Has f (Arr a Partial b)) r) m ->
          Exp c (RCons (Has f (Arr a Partial b)) r) n ->
          Exp c r {lub n m}
```

```
data Cls:: Set ~> *0 ~> Row HasType ~> *0 where
  Last:: (Pat s1 a g1 g2,Exp b g2 m) -> Cls s1 (Arr a m b) g1
  Next:: (Pat s1 a g1 g2,Exp b g2 m) ->
         Cls s (Arr a n b) g1 ->
         Cls {union s1 s} (Arr a {lub m n} b) g1
```

Note as in the `Lam` example we have used `(Row Hastype)` to track the shape of environments. Second, note the use the type functions `lub`, `allU`, and `union` to track termination behavior. The latter gives rise to a number of interesting narrowing problems. First lets see how the types of the constructors track both the type and the termination behavior of the term they construct. Consider:

```
--  (fn [] => 0 | (x:_) => x)

(Abs (Next (Pnil,(Const 0))
     (Last ((Pcons (Pvar `x) Pwild),(Var `x)))))

:: Exp (Arr [Int] Total Int) a Total
```

In the comment we give an ML-style lambda with multiple pattern matching clauses to describe the concrete syntax of the term. Next we give its actual construction in terms of the constructors of `Exp`, and finally we give its computed type. To compute this type, we need to solve several narrowing problems. First, study the types of the four components. The two patterns, and the two right-hand sides.

```
Pnil::                  Pat (Plus Univ Deny) [a] b b
Pcons (Pvar `x) Pwild:: Pat (Plus Deny (Times Univ Univ)) [c] d {`x:c; d}
Const 0::               Exp Int e Total
Var `x::                Exp f {`x:f; g} Total
```

The type rules for `Next` and `Last` indicate we need `union` the `Set`s of the two patterns. So we must narrow: `{union (Plus Univ Deny) (Plus Nil (Times Univ Univ))}` to get `(Plus Univ (Times Univ Univ))`. Next we must `lub` the termination behavior of the two right hand sides `{lub Total Total}` to get `Total`. So the sub term that is the argument to `Abs` has the type: `Cls (Plus Univ (Times Univ Univ)) (Arr [Int] Total Int) a`

By observing the type of `Abs` we see we must narrow `{allU (Plus Univ (Times Univ Univ))}` to get `Total`. This tells us that the patterns are exhaustive. Combined with the totality of the right-hand sides we narrow `{lub Total Total}` to get `Total`. Hence we arrive at the type of the term: `(Exp (Arr [Int] Total Int) a Total)`. The free type variable `a` tells us the term has this type in any environment (because it is a closed term with no free variables).

We can track non-termination from non-exhaustive patterns, the use of error, or recursion.

```
--  (fn [] => 0 | (x:(y:ys)) => x)
(Abs (Next (Pnil,(Const 0))
     (Last ((Pcons (Pvar `x) (Pcons (Pvar `y) (Pvar `ys))
:: Exp (Arr [Int] Partial Int) a b

-- (fn _ => error "oops")
(Abs (Last (Pwild,Error))) :: Exp (Arr a Partial b) c d

-- letrec f = (fn x => 1 + (f x))  in (f 3)
Let `f (abs `x (App (App (Oper Add) (Const 1)) (App (Shift (Var `f)) (Var `x))))
            (App (Var `f) (Const 3))
:: Exp Int g Partial
```

With this small language, every well-typed meta-level expression computes both the type and a termination behavior of the object program it represents. The computed termination is exact, except in the case of recursion where over approximates, declaring every partial function to be partial. One of the research

question is how to build such a model into the Ωmega implementation, and how to address the fact that it is too conservative for recursive functions. Our approach will be to assume all recursive programs are partial unless the user indicates they are not by giving a total function type in the functions prototype. Then we will incorporate more precise strategies for modelling the termination behavior of recursion[19, 6, 7].

11 Ambiguous types

Its possible to compute ambiguous solutions while narrowing. When we were exploring the design space for tracking termination, we initially declared the types of the variable and application constructor of `Exp` as follows.

```
data Exp:: *0 ~> Row HasType ~> Termination ~> *0 where
  Var::   Label a -> Exp t (RCons (Has a t) r) polyterm
  App::   Exp (Arr x m b) g n -> Exp x g p -> Exp b g {lub {lub p n} m}
  . . .
```

The idea was that we could make the termination behavior of a variable polymorphic (the `polyterm` type variable) rather than `Total`. After all a polymorphic termination could be `Total` if we needed it to be. Unfortunately, this under constrains the problem. For example consider declaring the term (in concrete syntax `(f y)`) to be partial.

```
ambig:: Exp Int (RCons (Has `f (Arr Int m Int)) (RCons (Has `y Int) RNil)) Partial
ambig = App (Var `f) (Shift (Var `y))
```

Here the partiality of the term could arise from either the `f` or the `y`. Typing this term requires solving the narrowing problem `{ {lub {lub a b} c} == Partial }`. But there are two solutions to this problem `{ a -> Total, b ->Partial }` and `{ a -> Partial }`. The type is ambiguous and is disallowed. In a context where either the `f` or the `y` is known (either `Total` or `Partial`), there is a unique solution. It is better to tightly constrain the problem. After all, in a strict language like Ωmega, every variable is total.

Another kind of ambiguity problem also sometimes arises when type checking. Consider narrowing `{ {lub x x} == x }`. This has two solutions `{ x -> Total }` and `{ x -> Partial }`. In some sense, since these two bindings exhaustively cover the space of possible bindings for `x`, the identity substitution (binding no variables) is also a solution. In some sense the ambiguity introduced by narrowing in this example is spurious.

We can also have spurious ambiguity in an infinite set of solutions. Consider narrowing `{ {plus n #0} == n }`. This is true for all `n`, and narrowing finds an infinite set of bindings `{ x -> #0 }`, `{ x -> #1 }`, `{ x -> #2 }` etc. But, just as in the previous example this set of bindings exhaustively covers the space of all possible bindings, so the identity substitution is also a solution. It would be good to recognize spurious ambiguity when it occurs, and remove it. The is an area for future research.

12 Is narrowing effective?

Narrowing has often been legitimately criticized for two reasons. (1) Problems for which narrowing is appropriate involve search. If the search space is too large, the performance of narrowing degrades. The search space becomes unmanageable when the number of variables becomes too large, and when the search space is not effectively pruned by constraints on the variables. (2) Narrowing is not applicable to computations over non-algebraically defined data (such as arithmetic over floating point numbers). Neither of these criticisms hold in the context of using narrowing to answer type-checking questions. Type-checking questions are always over inductively defined algebraic data, and the number of free variables in a type-checking question is related to the size of the program being type checked. People do not write programs that generate large narrowing problems. Typical questions have only a handful of variables, well within the range of effective solution using narrowing.

The problem of spurious ambiguity is also amenable to solution. In the paper, *Narrowing the Narrowing Space*[2], Antoy and Ariola identify the same problem and propose an ingenious solution. By memoizing previous narrowing steps, narrowing can recognize instances of a previous goals. The algorithm can build a graph representation of the narrowing space. They show that such a graph is a finite state automaton, and hence can be reformulated as a regular expression. In regular expression form, the spurious ambiguous solutions are recognizable and can be removed. Antoy reports that he knows of no implementation of this result. We are evaluating whether to build one.

Narrowing is a powerful technique whose power we have just begun to explore. It seems to provide just the right expressive power for writing functions at the type level, and it has a well studied theory upon which we can rely. In addition, it also allows programmers to use a uniform means of expressing computation at all levels – the writing of recursive equations.

References

[1] S. Antoy. Definitional trees. In H. Kirchner and G. Levi, editors, *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pages 143–157, Volterra, Italy, September 1992. Springer LNCS 632.

[2] S. Antoy and Z. M. Ariola. Narrowing the narrowing space. *Lecture Notes in Computer Science*, 1292:1–??, 1997. 9th International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP'97).

[3] Sergio Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 40(1):875–903, July 2005.

[4] Lennart Augustsson. Cayenne — a language with dependent types. *ACM SIGPLAN Notices*, 34(1):239–250, January 1999.

[5] Lennart Augustsson. Equality proofs in cayenne, July 11 2000.

[6] Robert S. Boyer and J. Strother Moore. *A Computational Logic*. Academic Press, New York, 1979.

[7] Jürgen Brauburger. *Automatic Termination Analysis for Functional and Imperative Programs*. PhD thesis, Technische Universität Darmstadt, 1999.

[8] Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *ACMCS*, 17(4):471–522, December 1985.

[9] Chen and Xi. Combining programming with theorem proving. In *Proceedings of the 2005 ACM/SIGPLAN International Conference on Functional Programming (ICFP'05)*, pages 66 – 77, September 2005.

[10] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *ICFP 2005*, 2005. http://www.cs.bu.edu/ hwxi/.

[11] Catarina Coquand. Agda is a system for incrementally developing proofs and programs. Web page describing AGDA:
`http://www.cs.chalmers.se/~catarina/agda/` .

[12] T. Coquand and P. Dybjer. Inductive definitions and type theory an introduction (preliminary version). *Lecture Notes in Computer Science*, 880:60–76, 1994.

[13] Karl Crary, Stephanie Weirich, and J. Gregory Morrisett. Intensional polymorphism in type-erasure semantics. In *ICFP*, pages 301–312, 1998.

[14] Rowan Davies. A refinement-type checker for Standard ML. In *International Conference on Algebraic Methodology and Software Technology*, volume 1349 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[15] P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. *Lecture Notes in Computer Science*, 1581:129–146, 1999.

[16] Michael Hanus and Ramin Sadre. An abstract machine for curry and its concurrent implementation in java. *Journal of Functional and Logic Programming*, 1999(Special Issue 1), 1999.

[17] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at `http://www.informatik.uni-kiel.de/~curry`, March 28, 2006.

[18] Simon Peyton Jones and Mark Shields. Practical type inference for arbitrary-rank types. Technical report, Microsoft Research, "December" 2003. `http://research.microsoft.com/Users/simonpj/papers/putting/index.htm`.

[19] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *POPL*, pages 81–92, 2001.

[20] Zhaohui Luo and Robert Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, LFCS, Computer Science Dept., University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, May 1992. Updated version.

[21] Connor McBride. Epigram: Practical programming with dependent types. In *Notes from the 5th International Summer School on Advanced Functional Programming*, August 2004. Available at: `http://www.dur.ac.uk/CARG/epigram/epigram-afpnotes.pdf` .

[22] Bengt Nordstrom. The ALF proof editor, March 20 1996.

[23] Emir Pasalic. *The Role of Type Equality in Meta-programming*. PhD thesis, OGI School of Science & Engineering at OHSU, October 2004. Available from: http://www.cs.rice.edu/~pasalic/thesis/body.pdf.

[24] Emir Pasalic and Nathan Linger. Meta-programming with typed object-language representations. In *Generative Programming and Component Engineering (GPCE'04)*, pages 136 – 167, October 2004. LNCS volume 3286.

[25] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[26] Frank Pfenning and Carsten Schürmann. *Twelf User's Guide*, 1.2 edition, September 1998. Available as Technical Report CMU-CS-98-173, Carnegie Mellon University.

[27] Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *LNAI*, pages 202–206, Berlin, July 7–10, 1999. Springer-Verlag.

[28] Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. A type system for certified binaries. *ACM SIGPLAN Notices*, 37(1):217–232, January 2002.

[29] Tim Sheard. Languages of the future. *Onward Track, OOPSLA'04. Reprinted in: ACM SIGPLAN Notices, Dec 2004*, 39(10):116–119, October 2004.

[30] Tim Sheard. Playing with types. Technical report, Portland State University, 2005. `http://www.cs.pdx.edu/~sheard`.

[31] Tim Sheard. Putting Curry-Howard to work. In *Proceedings of the ACM SIGPLAN 2005 Haskell Workshop*, pages 74–85, 2005.

[32] Tim Sheard, James Hook, and Nathan Linger. GADTs + extensible kind system = dependent programming. Technical report, Portland State University, 2005. `http://www.cs.pdx.edu/~sheard`.

[33] Tim Sheard and Nathan Linger. Programming with static invariants in omega, September 2004. Available from:
`http://www.cs.pdx.edu/~sheard/` .

[34] Tim Sheard and Emir Pasalic. Meta-programming with built-in type equality. In *Logical Frameworks and Meta-Languages workshop*, July 2004. Available at:
`http://cs-www.cs.yale.edu/homes/carsten/lfm04/`.

[35] Christopher A. Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *Conference Record of POPL'00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 214–227, Boston, Massachusetts, January 19–21, 2000.

[36] Peter J. Stuckey and Martin Sulzmann. Type inference for guarded recursive data types, February 2005. Available from:
`http://www.comp.nus.edu.sg/~sulzmann/` .

[37] Aaron Stump. Imperative LF meta-programming. In *Logical Frameworks and Meta-Languages workshop*, July 2004. Available at:
`http://cs-www.cs.yale.edu/homes/carsten/lfm04/`.

[38] Martin Sulzmann and Meng Wang. A systematic translation of guarded recursive data types to existential types, February 2005. Available from:
`http://www.comp.nus.edu.sg/~sulzmann/` .

[39] The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 7.4*. INRIA, 2003. http://pauillac.inria.fr/coq/doc/main.html.

[40] Edwin Westbrook, Aaron Stump, and Ian Wehrman. A language-based approach to functionally correct inperative programming. Technical report, Washington University in St. Louis, 2005. Available at:
`http://cl.cse.wustl.edu/`.

[41] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1997.

[42] Hongwei Xi. Applied type systems (extended abstract). In *In post-workshop proceedingds of TYPES 2003*, pages 394 – 408, 2004. Springer-Verlag LNCS 3085.

[43] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. *ACM SIGPLAN Notices*, 33(5):249–257, May 1998.

[44] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In ACM, editor, *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, ACM SIGPLAN Notices, pages 214–227, New York, NY, USA, 1999. ACM Press.

Context Dependent Procedures and Computed Types in ✓eriFun

Andreas Schlosser, Christoph Walther,
Michael Gonder, and Markus Aderhold

Fachgebiet Programmiermethodik, Technische Universität Darmstadt, Germany
http://www.informatik.tu-darmstadt.de/pm

Abstract

We present two enhancements of the functional language $\mathcal{L}$ which is used in the ✓eriFun system to write programs and formulate statements about them. *Context dependent procedures* allow to stipulate the context under which procedures are sensibly executed, thus avoiding runtime tests in program code as well as verification of absence of exceptions by proving stuck-freeness of procedure calls. *Computed types* lead to more compact code, increase the readability of programs, and make the well-known benefits of type systems available to non-freely generated data types as well. Since satisfaction of context requirements as well as type checking becomes undecidable, proof obligations are synthesized to be proved by the verifier at hand, thus supporting static code analysis. Information about the type hierarchy is utilized for increasing the performance and efficiency of the verifier.

1 Introduction

We develop the ✓eriFun system [1, 14, 15], an interactive system for the verification of statements about programs written in the functional first-order programming language $\mathcal{L}$ [12]. This language consists of definition principles for freely generated polymorphic data types, for procedures operating on these data types based on recursion, case analyses, let-expressions and functional composition, and for statements (called "lemmas" in $\mathcal{L}$) about the data types and the procedures. Procedures are evaluated in a call-by-value discipline. The data types *bool* with constructors *true* and *false*, and $\mathbb{N}$ for natural numbers with constructors 0 and $^+(\ldots)$ for the successor function are predefined in $\mathcal{L}$. Lemmas are defined by universal quantifications using case analyses and the truth values to represent connectives. Upon definition of a data type, each argument position of a constructor is provided with a selector function, e.g. $^-(\ldots)$ is the selector of constructor $^+(\ldots)$ thus representing the predecessor function, and *hd* and *tl* are the selectors of the *list*-constructor ::, cf. Fig. 1. Type variables are preceded by "@" and expressions of form $?cons(x)$ are used as shorthand notation for $x = cons(sel_1(x), \ldots, sel_n(x))$, where *cons* is a constructor and sel_i are the selectors belonging to *cons*, hence e.g. $?::(x)$ holds iff list x is not empty.

The language also supports *incomplete definitions* of procedures [16] (also called *loose specifications* or *underspecifications* in the literature), using the symbol $\star$ to denote an indetermined result in cases which usually cause a runtime error or an exception. For example, procedure !! of Fig. 1 computes the n^{th} element of a list, where list elements are addressed from left to right starting with 0. Hence the result of $(k\,!!\,n)$ is indetermined (denoted by $\star$ in the body of !!) if $|k| \leq n$. Incomplete definitions are also used to define *abstract mappings* like the arity function $\|\ldots\|$ of Fig. 3.

Figure 1 gives an example of an $\mathcal{L}$-program for searching in an ordered list by the *binary search method* as well as the lemmas stating soundness and completeness of the search procedure, cf. [13].

```
structure bool <= true, false
structure ℕ <= 0, ⁺(⁻:ℕ)
structure list[@X] <= ε, [infix] :: (hd:@X, tl:list[@X])

function [outfix] | (k:list[@X]):ℕ <=
if ?ε(k) then 0 else ⁺(| tl(k) |) end

function [infix] !!(k:list[@X], n:ℕ):@X <=
if ?ε(k) then ⋆ else if ?0(n) then hd(k) else (tl(k) !! ⁻(n)) end end

function ordered(k:list[ℕ]):bool <=
if ?ε(k)
  then true
  else if ?ε(tl(k))
       then true
       else if hd(k) > hd(tl(k)) then false else ordered(tl(k)) end
end end

function find(key:ℕ, a:list[ℕ], i, j:ℕ):bool <=
if | a | > j
  then if j = i
    then key = (a !! i)
    else if j > i
      then let h := i + ⌈½(j − i)⌉ in
           let mid := (a !! h) in
             if mid > key
               then find(key, a, i, ⁻(h))
               else if key > mid then find(key, a, ⁺(h), j) else true end
             end end end
      else false end end
  else false
end

function binsearch(key:ℕ, a:list[ℕ]):bool <=
if ?ε(a) then false else find(key, a, 0, ⁻(| a |)) end

lemma binsearch is sound <= ∀a:list[ℕ], key:ℕ
  if{binsearch(key, a), key ∈ a, true}

lemma binsearch is complete <= ∀a:list[ℕ], key:ℕ
  if{ordered(a), if{key ∈ a, binsearch(key, a), true}, true}
```

Figure 1: Searching a list by *binary search*

2 Context Dependent Procedures

2.1 Context Clauses

We call procedures which may only be executed in a certain context *context dependent*. Context dependency is an "old theme" in computer science. Almost any programming language provides some mechanism for stipulating context dependency, assembly languages, early dialects of LISP etc. being the rare exceptions. Nowadays, programming languages provide elaborate type systems which allow to check context properties upon compile time. Our focus when talking about context dependency are requirements which—differently from type checking—usually are undecidable so that theorem proving is needed to check satisfaction of the context constraints.

Generally, the context requirement for a procedure is a predicate over the formal parameters of the procedure. If this requirement is not satisfied in a calling context, the result of the procedure call—if any—may be arbitrary. Consequently, satisfaction of the context requirement in the calling context is a necessary prerequisite that the procedure behaves in the intended way.

A *context dependent procedure* f of an $\mathcal{L}$-program P is defined by expressions of the form

$$\texttt{function}\ f(x_1{:}\tau_1,\ldots,x_n{:}\tau_n){:}\tau <= \texttt{assume}\ c_f; body_f \quad . \tag{1}$$

The *context clause* $c_f \in \mathcal{T}(\Sigma(P),\{x_1,\ldots,x_n\})_{bool}$ given by the `assume` declaration is represented by a boolean term built with the function symbols (except f) defined by the data type and procedure definitions of program P (given by the signature $\Sigma(P)$ of P) and the formal parameters x_i of procedure f. A context clause c_f defines a precondition which needs to hold when executing procedure calls of f. Since the `assume` declaration is optional, *true* is used by default for c_f if no context clause is explicitly provided in the procedure body.

The semantics of a context dependent procedure is simply defined as the semantics of its *relativized* version $\texttt{function}\ f(x_1{:}\tau_1,\ldots,x_n{:}\tau_n){:}\tau <= body_f^{ctx}$, where $body_f^{ctx} := \mathit{if}\ c_f\ \mathit{then}\ body_f\ \mathit{else}\ \star\ \mathit{end}$ denotes the *relativized body* of f. Since all selectors are incompletely defined, selectors are assigned a context clause, too: For a constructor *cons* with the corresponding selectors $sel_1,\ldots,sel_n$, the context clause c_{sel_i} of a selector call $sel_i(x)$ is defined as $?cons(x)$ stating that a selector must only be applied to the constructor it belongs to.

For instance, procedure !! of Fig. 1 can be reformulated using a context clause as displayed in Fig. 2. This context clause demands that procedure calls $(k\,!!\,n)$ only appear in contexts which guarantee that n is a legal address of list k. As $|k| > n$ entails $k \neq \varepsilon$, the test for $?\varepsilon(k)$ (and the indetermined result $\star$ in turn) is omitted in the body of the context dependent version of procedure !!, hence the absence of exceptions when calling !! now is guaranteed *statically* by the context clause. Consequently, execution of procedure !! is *more efficient*, as the $\min(|k|,n)$ tests for $?\varepsilon(k)$ are saved in the context dependent version.

The definition of procedure *find* is refined as well in Fig. 2. The context clause demands $|a| > j \geq i$ for the upper bound j and the lower bound i of the search interval in list a. Also here, a more efficient procedure is obtained as tests performed dynamically upon each (recursive) call of procedure *find* are replaced by a static test: The tests for $|a| > j$ and $j > i$ are saved, both of which cause costs proportional to $\lceil \log_2(j-i+1) \rceil$ in the original version of find.

2.2 Context Hypotheses

To guarantee that procedures (or selectors) f are called in a valid context within a term t only, a so-called *context requirement* is generated for t expressing that the context under which any f is called in t entails the context clause c_f of f (with formal parameters in c_f replaced by the actual parameters of the call).

Definition 2.1 (Context Requirements). For a term t, the set $Octx(t) \subseteq Occ(t)$ is the set of all *context sensitive occurrences* in t, i.e. occurrences $\pi \in Occ(t)$ such that $t_{|\pi} = g(t_1,\ldots,t_n)$ and g is a selector or a procedure function symbol.[1]

[1] As usual, $Occ(t)$ is the set of all *occurrences* of t, $t_{|\pi}$ denotes the subterm of t at occurrence π, and term $t\,[\pi \leftarrow r]$ is obtained from t by replacing $t_{|\pi}$ by r.

The *context requirement* $CR(t)$ of term t is given as $AND(\{CR(t,\pi)|\pi \in Octx(t)\})$, where $CR(t,\pi) = if\{COND(t,\pi), \theta(c_g), true\}$ if $t_{|\pi} = g(t_1,\ldots,t_n)$, and θ replaces the formal parameters x_i of g by the actual parameters t_i.[2] ■

For a *procedure* f as in (1), the *context hypothesis* of f is defined as `lemma` $f\$ctx <= \forall x_1{:}\tau_1,\ldots,x_n{:}\tau_n\ CR(body_f^{ctx})$, and the *context hypothesis* of a lemma

$$\texttt{lemma}\ lem <= \forall x_1{:}\tau_1,\ldots,x_k{:}\tau_k\ body_{lem} \tag{2}$$

is defined as `lemma` $lem\$ctx <= \forall x_1{:}\tau_1,\ldots,x_k{:}\tau_k\ CR(body_{lem})$.

✓`eriFun` demands that the context hypothesis of a procedure f be verified *before* verification of lemmas and (other) procedures "calling" f starts. Also the context hypothesis of a lemma must be verified *before* verification of the lemma.

For example, using the context clauses of *hd*, *tl*, $^-(\ldots)$, and !!, context hypothesis !!$\$ctx$ is generated for the context dependent procedure !! of Fig. 2, and (using the context clauses of $^-(\ldots)$ and *find*) context hypothesis *binsearch*$\$ctx$ is generated for the context dependent version of procedure *binsearch*. The context hypothesis for lemma *binsearch is complete* of Fig. 1 is displayed in Fig. 2 as well.

2.3 Determination Hypotheses

When calling an *incompletely* defined procedure, so-called *stuck computations* may result. For each user-defined procedure `function` $f(x_1{:}\tau_1,\ldots,x_n{:}\tau_n){:}\tau <= \ldots$, ✓`eriFun` synthesizes a so-called *domain procedure* `function` $\nabla f(x_1{:}\tau_1,\ldots,x_n{:}\tau_n){:}bool <= \ldots$. Domain procedures ∇f are always completely defined—i.e. stuck computations never occur when calling ∇f—and provide an equivalent requirement for the absence of stuck computations when calling the "mother" procedure f, i.e. computation of $\nabla f(q_1,\ldots,q_n)$ yields *true* iff computation of $f(q_1,\ldots,q_n)$ does not get stuck, see [16] for details.

Figure 2 displays domain procedure ∇!! synthesized by ✓`eriFun` for procedure !!.[3] Hence absence of stuck computations when calling $(k\,!!\,n)$ is guaranteed iff computation of $\nabla!!(k,n)$ yields *true*. But as domain procedure ∇!! just provides a recursive definition for deciding $|k| > n$, computation of $(k\,!!\,n)$ does not get stuck iff $|k| > n$ is satisfied upon a procedure call $(k\,!!\,n)$.

Using domain procedures, absence of stuck computations can be determined *statically*. To this effect, a so-called *determination hypothesis* `lemma` $f\$det <= \forall x_1{:}\tau_1,\ldots,x_n{:}\tau_n\ if\{c_f, \nabla f(x_1,\ldots,x_n), true\}$ is generated for each context dependent procedure as given in (1). Determination hypotheses simply express that a procedure's *context clause* is *sufficient* for the *absence of stuck computations*. Hence the truth of a context requirement of a call of procedure f entails the absence of stuck computations for this call, provided f's determination hypothesis has been verified.[4] As an example, Fig. 2 displays the determination hypothesis !!$\$det$ generated for procedure !!.

3 Computed Types

3.1 Non-freely Generated Data Types

Data types in $\mathcal{L}$ are *freely generated*, which means that the semantics of *monomorphic* types τ of an $\mathcal{L}$-program P—either defined directly, such as *bool* and $\mathbb{N}$, or otherwise obtained as a monomorphic instance of a non-monomorphic data type, such as $list[\mathbb{N}]$, $list[list[\mathbb{N}]]$, etc.—can be defined as a free term algebra with carriers $\mathcal{T}(\Sigma(P)^c)_\tau$ over a signature $\Sigma(P)^c$ of constructor function symbols. However, often one is concerned

[2] $AND(\{b_1,\ldots,b_n\})$—sometimes also written as $AND(b_1,\ldots,b_n)$—abbreviates a boolean term representing the conjunction of the boolean terms b_i. $COND(t,\pi)$ is the conjunction of all conditions in t leading to subterm $t_{|\pi}$.

[3] When synthesizing ∇f for a context dependent procedure f, $body_f^{ctx}$ is to be used.

[4] Verification of determination hypotheses cannot generally be *demanded*, as calls of abstract mappings like procedure $\|\ldots\|$ in Fig. 3 are inherently indetermined.

```
function [infix] !!(k:list[@X], n:ℕ):@X <=
assume | k | > n; if ?0(n) then hd(k) else (tl(k) !! ⁻(n)) end

lemma !!$ctx <= ∀k:list[ℕ], n:ℕ
  if{| k | > n,
    if{?0(n), ?::(k), if{?::(k), if{?⁺(n), | tl(k) | > ⁻(n), false}, false}},
    true}

function ∇!!(k:list[@X], n:ℕ):bool <=
if | k | > n
  then if ?0(n)
       then ?::(k)
       else if ?::(k) then ∇!!(tl(k), ⁻(n)) else false end
     end
  else false
end

lemma !!$det <= ∀k:list[ℕ], n:ℕ if{| k | > n, ∇!!(k, n), true}

function find(key:ℕ, a:list[ℕ], i, j:ℕ):bool <=
assume if{i > j, false, | a | > j};
if j = i
  then key = (a !! i)
  else let h := i + ⌈½(j − i)⌉ in let mid := (a !! h) in
         if mid > key
           then find(key, a, i, ⁻(h))
           else if key > mid then find(key, a, h, j) else true end
       end end end
end

function binsearch(key:ℕ, a:list[ℕ]):bool <=
assume ordered(a); if ?ε(a) then false else find(key, a, 0, ⁻(| a |)) end

lemma binsearch$ctx <= ∀a:list[ℕ]
  if{ordered(a),
    if{?ε(a), true, if{?⁺(| a |), if{0 > ⁻(| a |), false, | a | > ⁻(| a |)}, false}},
    true}

lemma binsearch is complete$ctx <= ∀a:list[ℕ], key:ℕ
  if{ordered(a), if{key ∈ a, ordered(a), true}, true}
```

Figure 2: Context dependent procedures

```
structure pair[@X, @Y] <= [infix] • ([postfix] 1:@X, [postfix] 2:@Y)
structure vsym <= variable(v.index :ℕ)
structure csym <= constant(c.index :ℕ)
structure fsym <= function(f.index :ℕ)
structure pterm <= var(vbl:vsym), const(cst:csym),
                   apply(func:fsym, argument:pterm), pack(left:pterm, right:pterm)

function [outfix] ‖ (f:fsym):ℕ <= ⋆

function #(t:pterm):ℕ <= if ?pack(t) then #(left(t)) + #(right(t)) else 1 end

function wft(t:pterm):bool <=
case t of
  apply : if ‖ func(t) ‖ = #(argument(t)) then wft(argument(t)) else false end
  pack : if ?pack(left(t))
           then false
           else if wft(left(t)) then wft(right(t)) else false end
         end
  other : true
end

function wfgt(t:wft):bool <=
case t of
  apply : wfgt(argument(t))
  pack : if wfgt(left(t)) then wfgt(right(t)) else false end
  other : ?const(t)
end

function wfgtu(t:wfgt):bool <=
if ?apply(x) then ‖ argument(t) ‖ = 1 else false end

function wfsubst(σ:list[pair[vsym, wft]]):bool <=
if ?ε(σ)
  then true
  else if ?pack((hd(σ))2) then false else wfsubst(tl(σ)) end
end

function list$pair$vsym$wft(x:list[pair[vsym, pterm]]):bool <=
if ?ε(x)
  then true
  else if wft((hd(x))2) then list$pair$vsym$wft(tl(x)) else false end
end
```

Figure 3: Computed types in the *pterm* domain

with proper subsets of $\mathcal{T}(\Sigma(P)^c)_\tau$ only. For example, the set of prime numbers $\mathbb{P} \subset \mathbb{N}$ is used when working in number theory, and lists of "type" $list[\mathbb{P}]$ are returned by a procedure doing prime factorization. Also *normal forms* are used quite frequently, for example for efficient search: Procedure *binsearch* of Fig. 1, e.g., operates on ordered lists, where *ordered* is a proper subset of the freely generated data type $list[\mathbb{N}]$.

All these examples have in common that certain structures s one is concerned with—prime numbers, normal forms, etc.—are not freely generated but rather are proper subsets of some freely generated data types τ. Consequently, "recognizer procedures" `function` $s(x{:}\tau){:}bool <= \ldots$ must be used to decide whether an item of type τ is a member of s. Subsequently, we shall use such procedures like freely generated data types, and we call them *computed types*, as algorithmic definitions are provided for them.

Definition 3.1 (Computed Types). Let $\mathcal{P}$ be the set of all *polymorphic types* of an $\mathcal{L}$-program P, let $\mathcal{W} \subseteq \mathcal{P}$ be the set of *type variables*, and let $\mathcal{M} \subseteq \mathcal{P}$ be the set of all *monomorphic types*.[5] We extend the type system of P by a set $\mathsf{M} := \bigcup_{i\in\mathbb{N}} \mathsf{M}_i$ where $\mathsf{M}_0 := \mathcal{M}$ and $\mathsf{M}_{i+1} := \mathsf{M}_i \cup \{\gamma \in \Sigma(P) \mid \gamma : \delta \to bool \text{ for some } \delta \in \mathsf{M}_i\} \cup \{\zeta[\gamma_1, \ldots, \gamma_n] \mid \zeta \text{ is an } n\text{-ary type constructor and } \gamma_1, \ldots, \gamma_n \in \mathsf{M}_i\}$. P is the set of all polymorphic types where instantiation of type variables with computed types from $\mathsf{M} \setminus \mathcal{M}$ is allowed (but not necessary). The *base type* $\mathcal{P}(\gamma) \in \mathcal{P}$ of a type $\gamma \in \mathsf{P}$ is defined as γ if $\gamma \in \mathcal{M} \cup \mathcal{W}$, $\mathcal{P}(\gamma) := \mathcal{P}(\delta)$ if $\gamma \in \mathsf{M}_{i+1}$ and $\gamma : \delta \to bool$ for some $\delta \in \mathsf{M}_i$, and $\mathcal{P}(\zeta[\gamma_1, \ldots, \gamma_n]) := \zeta[\mathcal{P}(\gamma_1), \ldots, \mathcal{P}(\gamma_n)]$.

The *$\mathcal{P}$-subtype relation* $\leqslant_\mathcal{P}$ is defined as the reflexive and transitive closure of the *direct $\mathcal{P}$-subtype relation* $\lessdot_\mathcal{P} \subseteq \mathcal{P} \times \mathcal{P}$, which is defined as the smallest relation satisfying

1. $\xi(\tau) \lessdot_\mathcal{P} \tau$, if $\xi = \{@v_i/\zeta[@w_1, \ldots, @w_n]\}$ for some $@v_i \in \mathcal{W}(\tau)$, $@w_1, \ldots, @w_n \notin \mathcal{W}(\tau)$ where $@w_1, \ldots, @w_n$ are pairwise different and ζ is an n-ary type constructor,[6] and
2. $\xi(\tau) \lessdot_\mathcal{P} \tau$, if $\xi = \{@v_i/@v, @v_j/@v\}$ where $@v_i, @v_j \in \mathcal{W}(\tau)$, $@v \notin \mathcal{W}(\tau)$.

The *M-subtype relation* $\leqslant_\mathsf{M}$ is defined as the reflexive and transitive closure of the *direct M-subtype relation* $\lessdot_\mathsf{M} \subseteq \mathsf{P} \times \mathsf{P}$, which is defined as the smallest relation satisfying

3. $\gamma \lessdot_\mathsf{M} \delta$, if $\gamma \in \mathsf{M}_{i+1}$ and $\gamma : \delta \to bool$ for some $\delta \in \mathsf{M}_i$ and
4. $\zeta[\gamma_1, \ldots, \gamma_n] \lessdot_\mathsf{M} \zeta[\delta_1, \ldots, \delta_n]$, if $\gamma_j \lessdot_\mathsf{M} \delta_j$ for some $j \in \{1, \ldots, n\}$, $\gamma_k = \delta_k$ for each $k \in \{1, \ldots, n\} \setminus \{j\}$, and ζ is an n-ary type constructor.

The *subtype relation* $\leqslant_\mathsf{P} := \leqslant_\mathsf{M} \circ \leqslant_\mathcal{P} \subseteq \mathsf{P} \times \mathsf{P}$ is defined as the composition of the M-subtype relation $\leqslant_\mathsf{M}$ and the $\mathcal{P}$-subtype relation $\leqslant_\mathcal{P}$. ■

The subtype relation $\leqslant_\mathsf{P}$ defines a join-semilattice $(\mathsf{P}, \leqslant_\mathsf{P})$ on the set of polymorphic and computed types (modulo α-conversion, i.e. type variable renaming). Hence, for $\tau_1, \tau_2 \in \mathsf{P}$ the most special common super type is uniquely determined as the least upper bound $\tau_1 \sqcup \tau_2$ of types τ_1 and τ_2, i.e. $\tau_1, \tau_2 \leqslant_\mathsf{P} \tau_1 \sqcup \tau_2 \leqslant_\mathsf{P} \tau$ for each τ with $\tau_1, \tau_2 \leqslant_\mathsf{P} \tau$.

Figure 3 shows some computed types in the domain of the (freely generated) data type *pterm*.[7] Procedures $wft \in \mathsf{M}_1$, $wfgt \in \mathsf{M}_2$ and $wfgtu \in \mathsf{M}_3$ (meaning "well-formed term", "well-formed ground term" and "well-formed ground unit application") define computed types, where $\mathcal{P}(wfgtu) = \mathcal{P}(wfgt) = \mathcal{P}(wft) = pterm$ and $wfgtu \lessdot_\mathsf{M} wfgt \lessdot_\mathsf{M} wft \lessdot_\mathsf{M} pterm \lessdot_\mathcal{P} @a$. Since $pair[vsym, wft] \in \mathsf{M}_2$, $list[pair[vsym, wft]] \in \mathsf{M}_3$, and $wfsubst \in \mathsf{M}_4$, where $pair[vsym, wft] \lessdot_\mathsf{M} pair[vsym, pterm]$, the type hierarchy $wfsubst \lessdot_\mathsf{M} list[pair[vsym, wft]] \lessdot_\mathsf{M} list[pair[vsym, pterm]] \lessdot^+_\mathcal{P} list[pair[@x, @y]] \lessdot_\mathcal{P} list[@z] \lessdot_\mathcal{P} @q$ can be established for *wfsubst*, and $\mathcal{P}(wfsubst) = \mathcal{P}(list[pair[vsym, wft]]) = list[pair[vsym, pterm]]$.

[5] $\mathcal{L}$ uses *parametric* polymorphism [3]. Since $\mathbb{N}$ is predefined in $\mathcal{L}$, $\mathcal{M}$ is not empty.

[6] $\mathcal{W}(\tau)$ denotes the set of all type variables used in type τ.

[7] Data type *pterm* defines *packed terms* unifying terms and termlists into one structure. E.g., term F(G(A,B),C,D) is represented by $apply(\text{F}, pack(apply(\text{G}, pack(\text{A,B})), pack(\text{C,D})))$.

```
function binsearch(key:ℕ, a:ordered):bool <=
if ?ε(a) then false else find(key, a, 0, ⁻(| a |)) end

lemma binsearch is complete <= ∀a:ordered, key:ℕ
  if{key ∈ a, binsearch(key, a), true}

lemma binsearch is complete$ctx <= ∀a:ordered, key:ℕ
  if{key ∈ a, ordered(a), true}
```

Figure 4: Binary search with computed types

3.2 Type Checking with Computed Types

Since type checking becomes undecidable with the involvement of computed types, it has to be supported by theorem proving. To be able to check whether some term is of type $\zeta[\gamma_1, \ldots, \gamma_n] \in \mathsf{P} \setminus \mathcal{P}$, a so called *sort instance procedure* $\zeta\$\gamma_1\$\cdots\$\gamma_n : \mathcal{P}(\zeta[\gamma_1, \ldots, \gamma_n]) \to bool$ is synthesized which returns *true* iff its argument is of type $\zeta[\gamma_1, \ldots, \gamma_n]$. Sort instance procedures are defined recursively using the recursion structure of data type ζ. For example, procedure *list$pair$vsym$wft* is synthesized to check for all elements of its input list recursively if their second component is a well-formed term, cf. Fig. 3.

To check whether some term t has type $\gamma \in \mathsf{P}$, $t : \gamma$ for short, a boolean term $TC(t, \gamma)$ is created by defining

- $TC(t, \gamma) := true$, if $\gamma \in \mathcal{P}$ and $t : \gamma$, [8]
- $TC(t, \gamma) := if\{TC(t, \delta), \gamma(t), false\}$, if $\gamma \in \Sigma(P)$ and $\gamma \lessdot_{\mathsf{M}} \delta$, and
- $TC(t, \zeta[\gamma_1, \ldots, \gamma_n]) := if\{TC(t, \zeta[\delta_1, \ldots, \delta_n]), \zeta\$\gamma_1\$\cdots\$\gamma_n(t), false\}$, if ζ is an n-ary type constructor and $\zeta[\gamma_1, \ldots, \gamma_n] \lessdot_{\mathsf{M}} \zeta[\delta_1, \ldots, \delta_n]$.[9]

```
function apply.to.var(σ:wfsubst, v:vsym):wft <=
if ?ε(σ)
  then var(v)
  else if v = (hd(σ))_1 then (hd(σ))_2 else apply.to.var(tl(σ), v) end
end

function apply.to.term(σ:wfsubst, t:wft):wft <=
case t of
  var : apply.to.var(σ, vsym(t))
  const : t
  apply : apply(func(t), apply.to.term(σ, argument(t)))
  pack : pack(apply.to.term(σ, left(t)), apply.to.term(σ, right(t)))
end
```

Figure 5: Computed types as result types

For instance, for verifying $q : wfgtu$ wrt. the procedures of Fig. 3, $if\{wft(q), if\{wfgt(q), wfgtu(q), false\}, false\}$ is computed for $TC(q, wfgtu)$ (after *if*-terms have been normalized) and $if\{list\$pair\$vsym\$wft(\sigma), wfsubst(\sigma), false\}$ is the normalized result for $TC(\sigma, wfsubst)$.

[8] As $\gamma \in \mathcal{P}$ in this case, side condition "$t : \gamma$" is decided by "conventional" type checking.

[9] Type $\zeta[\delta_1, \ldots, \delta_n]$ is not uniquely determined by $\lessdot_{\mathsf{M}}$, but since the order in which the type hierarchy is traversed is irrelevant, an arbitrary direct super type of $\zeta[\gamma_1, \ldots, \gamma_n]$ is chosen.

Now to check for type correctness when calling a procedure as given in (1)—where, however, $\tau_i \in \mathsf{P}$ now is allowed—the boolean expression $AND(TC(x_1, \tau_1), \ldots, TC(x_n, \tau_n))$ is added conjunctively to the context clause c_f of procedure f. This guarantees implicitly that all actual parameters have the correct type in each procedure call whose context requirement is verified. Hence, computed types extend the parametric polymorphism of $\mathcal{L}$ by *inclusion* polymorphism [3]. The semantics of a procedure with computed types is defined as the semantics of its *relativized* version `function` $f(x_1{:}\mathcal{P}(\tau_1), \ldots, x_n{:}\mathcal{P}(\tau_n)){:}\tau$ <= *if* $AND(TC(x_1, \tau_1), \ldots, TC(x_n, \tau_n), c_f)$ *then* $body_f$ *else* $\star$ *end* . Similarly, lemmas as given in (2) can be defined using computed types by allowing $\tau_i \in \mathsf{P}$. Also here, the semantics of a lemma using computed types is defined as the semantics of its *relativized* version `lemma` *lem* <= $\forall x_1{:}\mathcal{P}(\tau_1), \ldots, x_n{:}\mathcal{P}(\tau_n)$ *if*$\{AND(TC(x_1, \tau_1), \ldots, TC(x_n, \tau_n)), body_{lem}, true\}$.

Figure 4 displays a refinement of procedure *binsearch* using procedure *ordered* of Fig. 1 as a computed type. Also lemma *binsearch is complete* uses the computed type *ordered*, thus allowing a more compact representation of the lemma. By the refinement of procedure *binsearch*, *ordered*(a) arises as an additional (trivial) proof obligation in the context hypothesis *binsearch is complete$ctx*.

Computed types are also allowed as *result types* of procedures, i.e. $\tau \in \mathsf{P}$ instead of $\tau \in \mathcal{P}$ may be allowed as well in procedures as given in (1). In such a case, a so-called *signature hypothesis* is synthesized as an $\mathcal{L}$-lemma `lemma` $f\$sig <= \forall x_1{:}\tau_1, \ldots, x_n{:}\tau_n\ TC(f(x_1, \ldots, x_n), \tau)$ in ✓`eriFun`. [10] The semantics of a procedure with a computed type τ as result type is simply defined as the semantics of a procedure which has its base type $\mathcal{P}(\tau)$ as result type.

Procedures *apply.to.var* and *apply.to.term* of Fig. 5 are examples for procedures with computed result types. The signature hypotheses for these procedures simply express that *well-formed* terms are obtained if *well-formed* substitutions are applied to *variables* or *well-formed* terms respectively, where the signature hypothesis generated for *apply.to.var* is required to verify the signature hypothesis for *apply.to.term*.

4 Reasoning with Computed Types

So far, the use of computed types only leads to more compact and readable definitions of procedures and lemmas in $\mathcal{L}$. This means that some syntactic sugar has been spread on our programming language for easing its use. Verification comes into play only to support type checking when computed types are involved, viz. for proving context, determination and signature hypotheses.

However, the main value of computed types comes with utilization of type information upon reasoning, as this may support verification considerably. The situation is quite comparable with the benefit of conventional type systems which do not only support writing of more compact and more readable code, but allow to detect program faults at compile time rather than at runtime only.

4.1 The *HPL*-Calculus

Statements about procedures of an $\mathcal{L}$-program are formulated in ✓`eriFun` as $\mathcal{L}$-lemmas, i.e. expressions of the form given in (2). The proof of a lemma usually requires induction, the base and step formulas of which are represented by *sequents* of the form $h_1, \ldots, h_n;\ \forall ih_1, \ldots, \forall ih_m \Vdash goal$ where $\{h_1, \ldots, h_n\}$ denotes the set of *hypotheses* defining the base or step case respectively. The set of *induction hypotheses* of a step case is given by $\{\forall ih_1, \ldots, \forall ih_m\}$, where the non-induction variables are universally quantified, and *goal*, called the *goalterm* of the sequent, represents the induction conclusion. The induction hypotheses and the goalterm are boolean terms, and the hypotheses are literals.[11]

The set of sequents defines the language of the *HPL-calculus* (abbreviating *Hypotheses*, *Programs* and *Lemmas*), which is the calculus in which the lemmas are proved. The application of a proof rule of this calculus to a sequent yields a finite set of sequents, which are obtained by altering the set of hypotheses, the

[10] Usually, signature hypotheses have a straightforward proof by induction according to the recursion structure of procedure f. Hence termination of f is verified before verification of the signature hypothesis begins.

[11] An atom a is an *if*-free boolean term, and a literal is an atom or a negated atom written as $\neg a$ or as $t \neq r$ for negated equations $t = r$. $\overline{l}$ stands for the complement of a literal l and $\overline{C} := \{\overline{l} \mid l \in C\}$ for a clause C.

set of induction hypotheses or the goalterm of the sequent to which the proof rule has been applied. Each proof rule is *sound* in the sense that the truth of all resulting sequents entails the truth of the sequent to which the proof rule has been applied. A proof in the *HPL*-calculus is represented by a *prooftree*, the nodes of which are given by sequents. The root node of a prooftree for a lemma *lem* is given by the *initial sequent* $\Vdash body_{lem}$, and the successor nodes are given by the sequents resulting from a proof rule application to the father node sequent. A proof of lemma *lem* is obtained if a *closed* prooftree can be built for *lem*, i.e. a prooftree where each leaf is a sequent of the form $\ldots \Vdash true$.

The *HPL*-calculus provides a set of 15 proof rules to create prooftrees [8, 15]. For example, *Induction* creates the base and step sequents from an initial sequent wrt. some induction axiom, *Use Lemma* applies a lemma to a sequent, *Case Analysis* creates successor sequents by a case split, *Unfold Procedure* "opens up" a procedure call in a goalterm, etc. Some of these proof rules require a *substitution* or a *term* as input, and in order to ensure well-formedness of the resulting sequent, syntactical requirements for these inputs have to be checked. Hence, with the use of computed types, these requirements need to be updated.

To supply the *HPL*-calculus with type information about the used variables, sequents are extended by an additional set of *type hypotheses*. Thus, a sequent with type hypotheses has the form

$$th_1, \ldots, th_l;\ h_1, \ldots, h_n;\ \forall ih_1, \ldots, \forall ih_m \Vdash goal \tag{3}$$

where the type hypotheses are literals of form $\gamma(x_i)$ with $\gamma \in \mathsf{P}$, where $\gamma(x_i) := true$ is omitted if $\gamma \in \mathcal{P}$.[12] The initial sequent of a prooftree for a lemma *lem* as given in (2) is $\tau_1(x_1), \ldots, \tau_k(x_k);\ ;\ \Vdash body_{lem}$.

When applying the *Induction* rule for using induction upon variables $x_1 : \gamma_1, \ldots, x_k : \gamma_k$ of (possibly *computed*) types γ_i, an additional sequent

$$th_1, \ldots, th_l;\ h_1, \ldots, h_n; \Vdash AND\left(\bigcup_{j=1}^{m}\bigcup_{i=1}^{k}\{TC(t_{j,i}, \gamma_i)\}\right) \tag{4}$$

is created for each step sequent of the form (3), demanding *type correctness* for each substitution $\{x_1/t_{j,1}, \ldots, x_k/t_{j,k}\}$ used to build the induction hypothesis ih_j (under the hypotheses h_i defining the step case). For instance, when proving $\forall x{:}ordered\ goal$ by structural *list*-induction upon x, the sequents

$$ordered(x);\ ?\varepsilon(x); \Vdash goal \tag{5}$$

$$ordered(x);\ ?::(x);\ goal[x/tl(x)] \Vdash goal \tag{6}$$

$$ordered(x);\ ?::(x); \Vdash ordered(tl(x)) \tag{7}$$

are generated as successor sequents of the initial sequent, where sequents (5) and (6) are the usual base and step sequents coming with the *list*-induction, and (7) is the *additional sequent* coming with (4) for guaranteeing type correctness of the substitution used to form the induction hypothesis.

But if $\forall x{:}list.ev\ goal$ is to be proved instead for a computed type $list.ev$, where $list.ev(x)$ iff the length of list x is even, structural *list*-induction becomes unsound, as $tl(x) : list.ev$ is false if $x : list.ev$ and $?::(x)$ holds. As the additional step sequent $list.ev(x);\ ?::(x); \Vdash list.ev(tl(x))$ cannot be proved, the prooftree obtained by *list*-induction upon $x : list.ev$ cannot be closed, thus avoiding an unsound induction proof. If, however, $\forall x{:}list[@X]\ goal$ is considered, the additional step sequent $;\ ?::(x); \Vdash if\{true, true, true\}$ is obtained which trivially simplifies to *true*, thus imposing no restriction upon the *list*-induction.

Also additional successor sequents have to be generated for other *HPL*-proof rules which expect terms or substitutions as input in order to guarantee *type correctness* as well as *context correctness* (thus entailing *well-typedness* in particular, cf. Sect. 3.2) of the rule input. For example, when employing *Use Lemma* to apply an instance $\sigma(body_{lem})$ of an $\mathcal{L}$-lemma as given in (2) to a sequent's goalterm, the successor sequent $\ldots \Vdash goal[\pi \leftarrow if\{\sigma(body_{lem}), goal_{|\pi}, true\}]$ is obtained for a sequent as given in (3).[13] Substitution $\sigma = \{x_1/t_1, \ldots, x_k/t_k\}$ (which must be provided when calling *Use Lemma*) replaces the universally quantified

[12] If γ has the form $\zeta[\gamma_1, \ldots, \gamma_n]$ the corresponding type hypothesis is actually built with the sort instance procedure of γ.

[13] We write $\ldots \Vdash goal$ if a successor sequent inherits all hypotheses, all type and all induction hypotheses of the father sequent. $\pi \in Occ(goal)$ is a parameter of the *Use Lemma* rule, where π is restricted to occurrences allowing terms of type *bool* only.

variables $x_i : \tau_i$ of lemma *lem* by terms t_i which use variables of the sequent only, and ✓eriFun's parser is used to check whether each t_i is a well-formed term of type τ_i. Now when allowing *computed* types so that $\tau_i \in \mathsf{P}$ may hold as well, type checking has to be supported by theorem proving. To this effect, the system generates the proof obligation

$$wtc_\sigma = if\{COND(goal, \pi), AND\left(\bigcup_{i=1}^{k}\{CR(t_i), TC(t_i, \tau_i)\}\right), true\}$$

expressing that—under the context $COND(goal, \pi)$ of the lemma application—each t_i is a *context correct* term of *type* τ_i. The system then demands verification of proof obligation wtc_σ simply by providing father sequent (3) with an additional successor sequent $\ldots \Vdash wtc_\sigma$. As the truth of *all* successor sequents is demanded for the father sequent to hold, proof obligation wtc_σ must be verified in order to obtain a closed proof tree.

Finally, a new *HPL*-rule *Relativize* has to be provided in addition since some verification problems require making type information explicit: If $goal_{|\rho} : \gamma$ holds for the subterm of *goal* at a user-provided occurrence ρ, *Relativize* creates the successor sequent $\ldots \Vdash goal[\pi \leftarrow if\{TC(goal_{|\rho}, \gamma), goal_{|\pi}, true\}]$ for a sequent of the form (3), where π selects the smallest superterm of $goal_{|\rho}$ with type *bool*.

For instance, when proving the context requirement for the recursive call of procedure *apply.to.var* of Fig. 5, the sequent

$$wfsubst(\sigma); \ldots \Vdash if\{?\varepsilon(\sigma), true, wfsubst(tl(\sigma))\} \tag{8}$$

arises. In order to prove (8), it is necessary to unfold the procedure call $wfsubst(\sigma)$ which is *implicitly* provided by the type hypothesis. To this effect, *Relativize* applied to the type hypothesis of (8) creates a new successor sequent $\ldots \Vdash if\{wfsubst(\sigma), if\{?\varepsilon(\sigma), true, wfsubst(tl(\sigma))\}, true\}$. Now the procedure call occurs explicitly in the goal term, thus being available for unfolding now.

4.2 The Evaluation Calculus

Goalterms of a sequent (3) are simplified by so-called *computed HPL*-proof rules. For instance, *Simplification* rewrites the sequent's goalterm using the definitions of the data types and procedures, the hypotheses and the induction hypotheses of the sequent and the lemmas already verified. These rewrites are performed by *symbolic evaluation* which is defined by another calculus, called the *evaluation calculus*, cf. [8, 17]. The language of this calculus is given by the set $\mathcal{T}(\Sigma(P), \mathcal{V})$ of first-order terms, where $\Sigma(P)$ stands for the signature of the function symbols defined by an $\mathcal{L}$-program P, and $\mathcal{V}$ is a set of typed variables used in the sequents. The inference rules of the evaluation calculus, called *evaluation rules*, are of the form "$\frac{term}{term'}$, if COND", where COND stands for a side condition which must be satisfied for applying the evaluation rule. We write $term \vdash_{H,A} term'$ if $term'$ originates from *term* by an evaluation rule using a set H of literals containing at least the sequent's hypotheses h_i and type hypotheses th_k, and clauses from a finite set $A \subset \mathcal{CL}(\Sigma(P), \mathcal{V} \cup \mathcal{U})$ which represents the sequent's induction hypotheses ih_j as well as the verified lemmas of P and are built with skolemized typed variables from $\mathcal{V}$ and universally quantified variables from a set $\mathcal{U}$ of typed variables. Thus, the set A contains type information as well, in particular the proven signature and context hypotheses.

Symbolic evaluations are computed in ✓eriFun by the *Symbolic Evaluator*, i.e. an automated theorem prover which considers the evaluation rules in a fixed order. Similarly to the *HPL*-calculus, some of the evaluation rules have to be modified. However, differently to the *HPL*-calculus, these updates are not only required for guaranteeing context and type correctness, but to utilize the knowledge about computed types and the subtype relation $\leqslant_{\mathsf{P}}$ for improving performance of the *Symbolic Evaluator*.

The rule *Affirmative hypothesis* of Fig. 6 is extended such that it simplifies terms by considering $\leqslant_{\mathsf{P}}$. For example, $wft(t) \vdash_{H,A} true$ is obtained in one evaluation step for the computed types of Fig. 3 if $wfgtu(t) \in H$, instead of exploring the direct subtype relation $<_{\mathsf{M}}$ step by step as it would be required otherwise. In addition, the rule *Evaluate then part* is modified such that it extends the set H of hypotheses under consideration of the subtype relation. If a new type hypothesis $\gamma(t)$ is to be added to H, unnecessary type hypotheses $\delta(t)$ with $\gamma \leqslant_{\mathsf{P}} \delta$ are removed from H. This keeps the set H as small as possible.

Affirmative hypothesis

$$\frac{a}{true} \text{ , if } a \in H \text{ or } a = \delta(d), \gamma \leqslant_{\mathsf{P}} \delta \text{ and } \gamma(d) \in H$$

Evaluate then part

$$\frac{if\{a,b,c\}}{if\{a,b',c\}} \text{ , if } \begin{cases} b \vdash_{H',A} b' \\ \text{where } H' = H \cup \{a\} \setminus \{\gamma(d) \in H | \delta \leqslant_{\mathsf{P}} \gamma\} \\ \text{if } a = \delta(d) \text{ for some } \delta \in \mathsf{P} \setminus \mathcal{P} \\ \text{and } H' = H \cup \{a\} \text{ else} \end{cases}$$

Affirmative assumption

$$\frac{a}{true} \text{ , if } \begin{cases} a = \sigma_\xi(lit) \text{ and } lit' \vdash^+_{H \cup \{\neg a\},A} false \text{ for some } \sigma_\xi, \\ \text{some } D \in A, \text{ some } lit \in D, \\ \text{some } \theta_{\xi'} \text{ with } \mathcal{U}(\theta_{\xi'}(\sigma_\xi(D))) = \emptyset \\ \text{and all } lit' \in \theta_{\xi'}(\sigma_\xi(D \setminus \{lit\})) \end{cases}$$

Assumption replacement

$$\frac{t}{if\{TC(\sigma_\xi(r),\tau),\sigma_\xi(r),t\}} \text{ , if } \begin{cases} t = \sigma_\xi(l) \text{ for some } \sigma_\xi, \text{ some } D \in A, \\ \text{some } l \texttt{=>} r \in D, \text{ some } \theta_{\xi'} \\ \text{with } \theta_{\xi'}(\sigma_\xi(D \setminus \{l \texttt{=>} r\})) \subseteq \overline{H}, \text{ and } t : \tau \end{cases}$$

Figure 6: Evaluation rules using computed types

The evaluation rules for using clauses from the clause set A are modified as well to incorporate the subtype relation $\leqslant_{\mathsf{P}}$. Central to this modification is the use of $\leqslant_{\mathsf{P}}$ for matching: A pattern type $\tau_1 \in \mathsf{P}$ matches a target type $\tau_2 \in \mathsf{P}$ *modulo computed types*, iff a type substitution ξ with $\mathcal{P}(\xi(\tau_1)) = \mathcal{P}(\tau_2)$ and $\tau_2 \leqslant_{\mathsf{P}} \xi(\tau_1)$ exists. Substitution ξ is the matcher of τ_1 and τ_2, iff $\xi(\tau_1) \sqcup \tau_2 \leqslant_{\mathsf{P}} \xi'(\tau_1) \sqcup \tau_2$ for every type substitution ξ'. A pattern term t_1 matches a target term t_2 *modulo computed types* iff there exists a term substitution σ and a type substitution ξ (modulo computed types) such that $\sigma_\xi(t_1) = t_2$.

Now evaluation rule *Affirmative assumption* for using verified clauses from A is modified by incorporation of *matching modulo computed types*. Hence the universally quantified variables of clauses from A can be matched with terms t_i having $\leqslant_{\mathsf{P}}$-smaller types in the goalterm. The types of terms t_i are determined using the type hypotheses in H and the signature information in A. These reasoning steps are obviously sound, as each property which is true in the domain of type δ holds in the domain of computed type $\gamma \leqslant_{\mathsf{P}} \delta$ as well. The benefit of matching modulo types stems from the fact that reasoning about subtypes is shifted into the matching algorithm instead of performing the required reasoning steps explicitly by several proof steps using the evaluation rules to refute subtype predicates. The dual evaluation rule *Negative assumption* which replaces a redex by *false* if σ_ξ is a matcher for *lit* and $\neg a$ uses matching modulo computed types as well. Similarly, evaluation rule *Assumption replacement* for equality reasoning using oriented equations $l \texttt{=>} r$ is updated to use matching modulo computed types, thus yielding the same benefits as for the *Affirmative/Negative assumption* rules. By adding a local type condition it is ensured that the replaced term has the correct type in the context of the rule application.

5 Related Work

Context dependency has been investigated for verifying termination of loops in imperative programs [2, 4]. Loops are translated here into tail-recursive procedures for which context requirements—called *termination predicates*—are *synthesized* which are sufficient for the procedures' termination. A verifier then is used to prove that the program context (given by the properties of the program variables used in the loop) entails the synthesized termination predicate. Termination predicates can be expressed by the context clauses of our proposal, which, however, must be supplied explicitly to the program code, see e.g. requirement $j \geq i$ for procedure *find* of Fig. 2.

The functional programming language *Miranda* supports definition of non-free data types as subtypes of free data types by stipulation of so-called *laws* [9], i.e. data types are enhanced by rewrite rules transforming values into normal forms, e.g. lists into ordered lists. This approach guarantees that values are always rewritten into normal forms, but there are no restrictions imposed on functions operating on these types, since the normal form has not to be preserved by the functions, but is restored by the *laws* automatically. Hence, type correctness is not enforced by type checking but ensured by rewriting. This approach is limited to subtypes which represent normal forms. For instance, subtypes like *wft* of Fig. 3 cannot be handled with laws.

ACL2 supports subtyping by so-called *guards* [6]. Guards are predicates which are used to check the arguments of functions for type correctness, corresponding to computed types in our setting. Guards are only available for the definition of procedures. Subtypes can neither be used in lemma definitions nor as result types of procedures. The main benefit of guards in ACL2 is to verify absence of exceptions upon the execution of a COMMON LISP program, thus corresponding to the proofs of *determination hypotheses* in our proposal.

PVS represents types by sets, and *computed* types can be defined as subsets via predicates [7], thus providing a framework most comparable to our approach. Differently to our proposal, PVS supports *polymorphic* computed types as well *dependent types*, which merge context dependent procedures and computed types into a single concept. However, reasoning about computed types is always performed explicitly by verifying PVS proof obligations, whereas it is partially incorporated into the reasoning machinery in our proposal (thus utilizing well-known benefits from classical theorem proving, see [10, 11] and see [18] for an exhaustive account on subsequent developments).

6 Conclusion

We presented two enhancements—implemented in an experimental system [5]—of the functional language $\mathcal{L}$ which is used in the ✓`eriFun` system to write programs and formulate statements about them. *Context dependent procedures* allow to specify the context under which procedures are sensibly executed, thus avoiding runtime tests in program code, which would be required otherwise. The tests whether a procedure is called in a program environment which guarantees the procedure's context demand can be performed statically. Proof obligations are generated to be proved by the verification system at hand. Context dependent procedures are also used to verify absence of exceptions statically by proving stuck-freeness of procedure calls. *Computed types* lead to more compact code and increase readability of programs, and the well-known benefits of type systems in programming languages become available for non-freely generated data types as well. Information about the type hierarchy is utilized for increasing performance and efficiency of the verifier. As type-checking becomes undecidable, proof obligations are synthesized. To show type correctness and well-typedness of expressions, these proof obligations are proved by the verifier.

Context dependent procedures and computed types do not subsume each other: Using context dependent procedures only, restrictions on the return type of procedures cannot be formulated, which is accomplished by computed types. Using computed types only prevents the specification of execution contexts for procedures which cannot be expressed by unary predicates.

Presently, we are investigating various upgrades of our proposal: We intend to allow *composition* of computed types by the set-theoretic operators $\cap$ and $\cup$ to write, for example, `function` *prime factors*(n :

$\mathbb{N}) : \mathit{list}[\mathbb{P}] \cap \mathit{ordered} <= \ldots$. Also *dependent computed types* as provided by PVS are under investigation to define a computed type in dependence of procedure parameters or lemma variables respectively. For instance, using such a feature one may write for procedure *find* of Fig. 2 `function` $\mathit{find}(\mathit{key}{:}\mathbb{N}, a{:}\mathit{list}[\mathbb{N}] \cap |\_| > j, i{:}\mathbb{N}, j{:}\ \_ \geq i){:}\mathit{bool} <= \ldots$ where a is assigned the dependent type $\mathit{list}[\mathbb{N}] \cap |\_| > j$ and j is assigned the dependent type $\_ \geq i$, altogether expressing $|a| > j \geq i$. Whether such expressions increase readability of programs is a matter of taste (and of use, of course). As the requirements for the parameters a and j in the example can be expressed by context dependent procedures as well, cf. Fig. 2, the key question is here whether dependent computed types contribute to *automated* reasoning in the same way as computed types do. Finally, it has to be investigated whether *polymorphic* computed types as provided by PVS are a useful enhancement.

References

[1] http://www.verifun.org.

[2] J. Brauburger and J. Giesl. Approximating the Domains of Functional and Imperative Programs. *Science of Comp. Prog.*, 35:113–136, 1999.

[3] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[4] J. Giesl, C. Walther, and J. Brauburger. Termination Analysis for Functional Programs. In W. Bibel and P. Schmitt, editors, *Automated Deduction - A Basis for Applications, vol. 3*, pages 135–164. Kluwer Academic Publishers, 1998.

[5] M. Gonder. Entwurf und Implementierung kontextabhängiger Prozeduren und Untertypen in ✓`eriFun` . Diploma thesis, Programmiermethodik, TU Darmstadt, Mar. 2006.

[6] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, June 2000.

[7] J. Rushby, S. Owre, and N. Shankar. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE Trans. on Softw. Eng.*, 24(9):709–720, Sept. 1998.

[8] S. Schweitzer. *Symbolische Auswertung und Heuristiken zur Verifikation funktionaler Programme*. Doctoral dissertation, Programmiermethodik, TU Darmstadt, to appear.

[9] S. Thompson. Laws in Miranda. In *Proceedings of the 1986 ACM conference on LISP and functional programming (LFP)*, pages 1–12. ACM Press, 1986.

[10] C. Walther. *A Many-Sorted Calculus Based on Resolution and Paramodulation*. Research Notes in Artificial Intelligence. Pitman and Morgan Kaufmann, London and Los Altos, 1987.

[11] C. Walther. Many-Sorted Unification. *J. ACM*, 35(1):1–17, 1988.

[12] C. Walther, M. Aderhold, and A. Schlosser. The $\mathcal{L}$ 1.0 Primer. Technical Report VFR 06/01, Programmiermethodik, TU Darmstadt, Apr. 2006.

[13] C. Walther and S. Schweitzer. A Verification of Binary Search. Technical Report VFR 02/02, Programmiermethodik, TU Darmstadt, Feb. 2002.

[14] C. Walther and S. Schweitzer. About ✓`eriFun`. In F. Baader, editor, *19th International Conference on Automated Deduction (CADE)*, volume 2741 of *LNAI*, pages 322–327. Springer, 2003.

[15] C. Walther and S. Schweitzer. Verification in the Classroom. *Journal of Automated Reasoning*, 32(1):35–73, 2004.

[16] C. Walther and S. Schweitzer. Reasoning about Incompletely Defined Programs. In G. Sutcliffe and A. Voronkov, editors, *12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 3835 of *LNAI*, pages 427–442. Springer, 2005.

[17] C. Walther and S. Schweitzer. A Pragmatic Approach to Equality Reasoning. Technical Report VFR 06/02, Programmiermethodik, TU Darmstadt, May 2006.

[18] C. Weidenbach. Sorted Unification and Tree Automata. In W. Bibel and P. Schmitt, editors, *Automated Deduction - A Basis for Applications*, volume 1, pages 291–320. Kluwer Academic Publishers, Dordrecht, 1998.

Functional programming with higher-order abstract syntax and explicit substitutions

Brigitte Pientka
School of Computer Science
McGill University
Montreal, Canada
bpientka@cs.mcgill.ca

Abstract

This paper sketches a foundation for programming with higher-order abstract syntax and explicit substitutions based on contextual modal type theory [NPP05]. Contextual modal types not only allows us to cleanly separate the representation of data objects from computation, but allow us to recurse over data objects with free variables. In this paper, we extend these ideas even further by adding first-class contexts and substitutions so that a program can pass and access code with free variables and an explicit environment, and link them in a type-safe manner. We sketch the static and operational semantics of this language, and give several examples which illustrate these features.

1 Introduction

Higher-order abstract syntax is a simple well-recognized technique for implementing languages with variables and binders. This issue typically is key when implementing evaluators, compilers or automated reasoning systems. The central idea behind higher-order abstract syntax is to implement object variables and binders by variables and binders in the meta-language (i.e. functional programming language). One of the key benefits behind higher-order abstract syntax representations is that one can avoid implementing common and tricky routines dealing with variables, such as capture-avoiding substitution, renaming and fresh name generation.

Higher-order abstract syntax and its usefulness have long been demonstrated within the logical framework LF [HHP93] and its implementation in the Twelf system [PS99]. However it has been difficult to extend mainstream functional programming languages with direct support for higher-order syntax encodings. The difficulty is due to the fact that higher-order abstract syntax encodings are not inductive in the usual sense. The problem is that recursion over higher-order abstract syntax requires one to traverse a λ-abstraction and hence we need to be able to reason about "open" terms. Building on ideas in [NPP05], we present a novel approach based on contextual modal types which allows recursion over open terms and also supports first-class environments and contexts.

Most closely related to our work are previous proposals by Despeyroux, Pfenning and Schürmann [DPS97] where the authors present a modal λ-calculus which supports primitive recursion over higher-order encodings via an iterator. However, function definitions via iteration do not support pattern matching and do not easily support reasoning with dynamic assumption. This problem is addressed in [SPS05] and the ∇-calculus is proposed as a remedy. It also serves as a foundation for the *Elphin* language. Their work requires scope stacks and explicit operations on them such as popping an element of the scope stack. A function is then executed within a certain scope. The necessity of keeping track of the current scope also permeates the operational semantics, which explicitly keeps track of scope stacks.

In contrast to these approaches we believe that the contextual modal type theory [NPP05] can provide a clean and elegant framework for extending functional programming with support for higher-order abstract

syntax and pattern matching. Since we allow abstraction over context variables, we can associate a scope (context) with each argument passed to a function rather with the function itself. This allows us to write programs which accept open data objects as inputs and enforces stronger invariants about data objects and programs. We also believe this will facilitate the reasoning about the programs we write. Previous approaches by [SPS05] only allow open objects during recursion but require the objects to be closed at the beginning of the computation. Moreover, it is possible to support first-class substitutions which are independently interesting and useful. First-class substitutions allow for example programming with explicit environments, which has a wide range of applications such as an environment based interpreter, or constant elimination algorithm. Our underlying contextual modal type system guarantees that programs can pass and access open terms and that it can safely linked with an environment.

In this paper we take a first step towards designing a foundation for programming with higher-order abstract syntax and explicit substitution based on contextual modal type theory. We first introduce an example to highlight some of the issues when programming with higher-order abstract syntax and then present a theoretical foundation by adapting and extending contextual modal type theory. Finally, we give several examples illustrating the basic ideas of programming with explicit substitutions. Since explicit substitutions can be viewed as value-variable pairs, this facility essentially allows us to model explicit environments. Moreover, our underlying type system will ensure the correct usage of this environment and statically enforce crucial invariants such as every "free" variable occurring in a term M is bound in some environment.

2 Motivating example

In this section we briefly discuss some of our main ideas and concerns using a simple example which analyzes and compares the structure of lambda-terms. To add higher-order encodings to a functional programming language we follow the approach taken in [DPS97, SPS05] to separate the representation and computation level via a modal operator. Data objects M can make use of higher-order abstract syntax and are injected into programs via the box-construct $\mathsf{box}\ M$ where M denotes object-level data. The computation level describes our functional programs which operate on object-level terms. On this level, we allow recursion and pattern matching against object-level terms to extract sub-terms. Unlike previous proposals however, every object M carries its own local context Ψ such that $\mathsf{box}(\Psi.\, M)$ thereby allowing programming with open objects.

To illustrate consider first the following two definitions of lambda-terms based on higher-order abstract syntax.

Object-level expressions			Object-level terms		
lam	:	$(\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp}.$	lam'	:	$(\mathsf{term} \to \mathsf{term}) \to \mathsf{term}.$
app	:	$\mathsf{exp} \to \mathsf{exp} \to \mathsf{exp}.$	app'	:	$\mathsf{term} \to \mathsf{term} \to \mathsf{term}.$

We are interested in defining a function $\mathsf{related}$ which checks whether an expression is related to a term. This function just compares the basic shape of a term and an expression but it does not check for equality of variable names. For example, we consider the term $\mathsf{lam}\ \lambda x.\mathsf{lam}\ \lambda y.\mathsf{app}\ x\ y$ to be related to the expression $\mathsf{lam}'\ \lambda x.\mathsf{lam}'\ \lambda y.\mathsf{app}'\ y\ x$. Hence our computation would start of with

$$\mathsf{related}\ (\mathsf{box}(\cdot.\, \mathsf{lam}\ \lambda x.\mathsf{lam}\ \lambda y.\mathsf{app}\ x\ y))\ (\mathsf{box}(\cdot.\, \mathsf{lam}'\ \lambda x.\mathsf{lam}'\ \lambda y.\mathsf{app}'\ y\ x))$$

Recursively, we need to traverse the lambda-binder and in the next iteration we compute

$$\mathsf{related}\ (\mathsf{box}(x{:}\mathsf{term}.\, \mathsf{lam}\ \lambda y.\mathsf{app}\ x\ y)\ (\mathsf{box}(x{:}\mathsf{exp}.\, \mathsf{lam}'\lambda y.\ \mathsf{app}'\ y\ x))$$

As we see every argument carries its own local context, which is extended when we traverse the lambda-binder. In the next iteration, the local contexts are again extended.

$$\mathsf{related}\ (\mathsf{box}(x{:}\mathsf{term}, y{:}\mathsf{term}.\, \mathsf{app}\ x\ y))\ (\mathsf{box}(x{:}\mathsf{exp}, y{:}\mathsf{exp}.\, \mathsf{app}'\ y\ x))$$

Finally, we need to check

$$\mathsf{related}\ (\mathsf{box}(x{:}\mathsf{term}, y{:}\mathsf{term}.\, x))\ (\mathsf{box}(x{:}\mathsf{exp}, y{:}\mathsf{exp}.\, y))$$

How could one write a recursive function related and what type should it have? – The function related takes as input open object of type term and open object of type exp. The local context associated with each open object will be reflected in its type. While in previous proposals object-level data of type term was given the type □term, we write term$[x{:}\mathsf{term}]$ for an open object M which has type term in the context x:term. In other words the object M is well-typed in the context x:term. Similarly, we write exp$[x{:}\mathsf{exp}, y{:}\mathsf{exp}]$ for an open object M which has type exp in the context x:exp, y:exp. To actually write recursive programs and assign a type to it, we need to be able to abstract over the concrete context, since it changes during execution and we must characterize valid instances for context variables. Valid instances of context variables can be described by context schemas (= worlds) (see also [SP03]). Let exprW be the constant describing the context schema $x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp}$, and termW be the constant describing the context schema $y_1{:}\mathsf{term}, \ldots, y_k{:}\mathsf{term}$. Then we can declare a dependent type for the function related as :

$$\mathsf{related} : \Pi\gamma{:}\mathsf{exprW}.\Pi\psi{:}\mathsf{termW}.\mathsf{term}[\psi] \rightarrow \mathsf{exp}[\gamma] \rightarrow \mathit{bool}.$$

Before we present the code for the function related, we discuss a few considerations here. First, a recursive function will be defined via pattern matching and needs to extract sub-expression. To describe "holes" which can be matched against open sub-expressions, we draw upon ideas from [PP03, NPP05] and use contextual modal variables $u[\sigma]$. $u[\sigma]$ denotes a closure with the postponed substitution σ. Second, we need to be able to pattern match against object-level variables. This will be achieved by imposing constraints on occurrences of context variables. The context variable $\psi(x{:}\mathsf{term})$ denotes a context of type termW which must contain at least one element x:term. We can now give the code for the function related next.

```
fun related [g] [g'] (box g(x:term).x)  (box g'(y:exp).y) =
     true
  | related [g] [g']
            (box g. lam [x]. u[id_g, x/x])
            (box g'. lam' [y]. v[id_g',y/y']) =
    related [g, x:term] [g', y:exp]
             (box g, x:term. u[id_g, x/x])
             (box g', y:exp. v[id_g', y/y'])
  | related [g] [g']
            (box g. app u1[id_g] u2[id_g])
            (box g'. app' v1[id_g] v2[id_g]) =
     related [g] [g'] (box g. u1[id_g]) (box g'.v1[id_g'])
      andalso
     related [g] [g'] (box g. u2[id_g]) (box g'.v2[id_g'])
```

It is also worth pointing out that unlike previous proposals we do not require objects to be closed at the start of evaluation. We can easily ask whether $\mathsf{box}(x{:}\mathsf{term}.\, \mathsf{lam}\ \lambda y.x)$ is related to $\mathsf{box}(x{:}\mathsf{exp}.\, x)$ for example. The answer is obviously no.

An important issue in our setting is α-equivalence and the scope of context variables. Do we consider $\mathsf{box}(\psi(x{:}\mathsf{term}).\, x)$ α-equivalent to $\mathsf{box}(\psi(y{:}\mathsf{term}).\, y)$? Is $\mathsf{box}(x{:}\mathsf{term}, y{:}\mathsf{term}.\, y)$ α-equivalent to $\mathsf{box}(z{:}\mathsf{term}, w{:}\mathsf{term}.\, w)$? What operations do we allow on context variables? – In the following theoretical development, we will pay careful attention to these issues.

3 Formalities

Object-level terms and typing In this section, we introduce the formal definition and type system based on contextual modal type theory. We start with th object language which is defined following ideas in [NPP05]. For simplicity, we restrict it to the simply-typed fragment.

Types	A, B, C	::=	$\alpha \mid A \to B$
Normal Terms	M, N	::=	$\lambda x.\, M \mid R$
Neutral Terms	R	::=	$x \mid c \mid R\, N \mid u[\sigma]$
Substitutions	σ, ρ	::=	$\cdot \mid \sigma, M/x \mid s[\sigma] \mid \mathsf{id}_{\psi(\omega)}$
Contexts	Ψ, Φ	::=	$\cdot \mid \Psi, x{:}A \mid \psi(\omega)$
Meta-contexts	Δ	::=	$\cdot \mid \Delta, u{::}A[\Psi] \mid \Delta, s{::}\Psi[\Phi]$
Constraints	ω	::=	$\cdot \mid \omega, x{:}A$

There are several interesting aspects about the simply-typed modal lambda-calculus. First, we distinguish between ordinary bound variables x and contextual modal variables $u[\sigma]$. Contextual modal variable $u[\sigma]$ denotes a closure with the postponed substitution σ. As we briefly alluded to in the previous section, contextual modal variables will be used to define pattern matching. Sometimes we also call contextual modal variables meta-variables. Our intention is to apply σ as soon as we know which term u should stand for. The domain of σ describes the free variables which can possibly occur in the term which represents u. For more details on contextual modal variables we refer the interested reader to [NPP05]. Here we propose to extend the calculus with context variables ψ and substitution variables $s[\sigma]$. Context variables ψ may be annotated with constraints ω which impose condition on the context ψ. For example $\psi(x{:}A)$ denotes a context which contains the variable x of type A. We also note that $\psi(\omega), \Psi$ is only well-formed if the variables mentioned in ω are not also declared in Ψ. At the moment, we restrict the use of constraints to context variables occurring in object. Context variables occurring in types are not allowed to be associated with any constraints[1]. We also require a first-class notion of identity substitution $\mathsf{id}_{\psi(\omega)}$. Abstracting over contexts and substitution seems an interesting and essential next step, if we aim at using contextual modal types as a foundation for programming with higher-order abstract syntax.

We assume that types and context schemas are declared in a signature similar to type and world declarations in Twelf [SP03]. We typically suppress the signature Σ since it never changes during a typing derivation, but keep in mind that all typing judgments have access to a well-formed signature. Checking a type A is well-formed in a signature Σ is straightforward since there are no dependencies. Next, we follow essentially ideas in [NPP05] to describe object-level canonical forms only. We assume only simple types (no dependencies) and object level type constants a together with constants denoting context schemas have been declared in a signature. Next, we describe the main typing judgments and typing rules.

$\Delta; \Psi \vdash M \Leftarrow A$	Check normal object M against type A
$\Delta; \Psi \vdash R \Rightarrow A$	Synthesize type A for atomic object R
$\Delta; \Phi \vdash \sigma \Leftarrow \Psi$	Check substitution σ against context Ψ

We will tacitly rename bound variables, and maintain that contexts and substitutions declare no variable more than once. Note that substitutions σ are defined only on ordinary variables x and not modal variables u. We also streamline the calculus slightly by always substituting simultaneously for all ordinary variables. This is not essential, but saves some tedium in relating simultaneous and iterated substitution. We will omit here the definitions for well-formed contexts and well-formed constraints, but focus on typing of terms and substitutions. During typing we refer to $\omega : W$ which guarantees that the constraints ω correspond to the context schema W.

[1] Allowing constraints in types would yield to complications when defining substitution

Object-level terms

$$\frac{\Delta; \Psi, x{:}A \vdash M \Leftarrow B}{\Delta; \Psi \vdash \lambda x.M \Leftarrow A \to B} \qquad \frac{\Delta; \Psi \vdash R \Rightarrow P' \quad P' = P}{\Delta; \Psi \vdash R \Leftarrow P}$$

$$\frac{x{:}A \in \Psi}{\Delta; \Psi \vdash x \Rightarrow A} \qquad \frac{x{:}A \in \omega \quad x{:}A \notin \Psi \quad \psi{:}W \in \Delta \quad \omega : W}{\Delta; \psi(\omega), \Psi \vdash x \Rightarrow A} \qquad \frac{c{:}A \in \Sigma}{\Delta; \Psi \vdash c \Rightarrow A}$$

$$\frac{u{::}A[\Phi] \in \Delta \quad \Delta; \Psi \vdash \sigma \Leftarrow \Phi}{\Delta; \Psi \vdash u[\sigma] \Rightarrow A} \qquad \frac{\Delta; \Psi \vdash M \Rightarrow A \to B \quad \Delta; \Psi \vdash N \Leftarrow A}{\Delta; \Psi \vdash M\, N \Rightarrow B}$$

Object-level substitutions

$$\frac{}{\Delta; \Psi \vdash \cdot \Leftarrow \cdot} \qquad \frac{\Delta; \Psi \vdash \sigma \Leftarrow \Phi \quad \Delta; \Psi \vdash M \Leftarrow A}{\Delta; \Psi \vdash (\sigma, M/x) \Leftarrow (\Phi, x{:}A)}$$

$$\frac{s{::}\Phi_1[\Phi_2] \in \Delta \quad \Phi = \Phi_1 \quad \Delta; \Psi \vdash \rho \Leftarrow \Phi_2}{\Delta; \Psi \vdash (s[\rho]) \Leftarrow \Phi} \qquad \frac{\psi : W \in \Delta \quad \omega : W}{\Delta; \psi(\omega), \Psi \vdash \mathsf{id}_{\psi(\omega)} \Leftarrow \psi(\omega)}$$

We note that we require the usual conditions on bound variables. For example in the rule for context and lambda-abstraction the bound variable ψ and x resp. must be new and cannot already occur in the context Ψ. This can be always achieved via alpha-renaming.

Computation-level expressions Our goal is to cleanly separate the object level and the computation level. While the object level describes data, the computation level describes the programs which operate on data. Computation-level types may refer to object-level types via the contextual type $A[\Psi]$ which denotes an object of type A which may contain the variables specified in Ψ. To allow quantification over context variables ψ, we introduce a dependent type $\Pi\psi{:}W.\tau$ where W denotes a context schema.

Types	τ	::=	$A[\Psi] \mid \Phi[\Psi] \mid \tau_1 \to \tau_2 \mid \Pi\psi{:}W.\tau$
Expressions	e	::=	$y \mid \mathsf{rec}\ f.e \mid e_1\ e_2 \mid \mathsf{fn}\ y.e \mid \Lambda\psi.e \mid e\ \lceil\Psi\rceil \mid (e : \tau)$
			$\mid \mathsf{box}(\Psi.\, M) \mid \mathsf{sbox}(\Psi.\, \sigma) \mid \mathsf{case}\ e\ \mathsf{of}\ p_1 \mid \ldots \mid p_n$
Branch	p, q	::=	$\epsilon u{::}A[\Psi].p \mid \epsilon s{::}\Phi[\Psi].p \mid \mathsf{box}(\Psi.\, M) \mapsto e \mid \mathsf{sbox}(\Psi.\, \sigma) \mapsto e$
Contexts	Γ	::=	$\cdot \mid \Gamma, y{:}\tau$

Data can be injected into programs via the box-construct $\mathsf{box}(\Psi.\, M)$. Here M denotes an object-level term M which may contain the variables specified in the context Ψ. Similarly, we can inject substitutions $\mathsf{sbox}(\Psi.\, \sigma)$ where Ψ is the range of the substitution σ. Since substitutions can be viewed as pairs between variables and object-level terms, this facility essentially allows us to model explicit environments. Finally, we allow pattern matching on object-level terms via case-statement. To simplify the theoretical development, we require that all contextual modal variables occurring in a pattern are explicitly specified. However we do not yet consider matching against context variables. We overload the $\to$ which is used to denote function types on the object level as well as the computation level. Also we may use x and y for object-level variables and program variables. However, it should be clear from the usage which one we mean.

Next, we consider bidirectional typing rules for programs. We distinguish here between typing of expressions and patterns. Note that in order to type expressions and patterns we will refer to the typing of object-level terms.

$\Delta; \Gamma \vdash e \Leftarrow \tau$	check expression e against type τ
$\Delta; \Gamma \vdash e \Rightarrow \tau$	synthesize type τ for expression e
$\Delta; \Delta'; \Gamma \vdash p \Leftarrow (\tau' \to \tau)$	check branch p has type $\tau' \to \tau$

The typing rules for expressions are next.

Expressions

$$\frac{\Delta, \psi{:}W; \Gamma \vdash e \Leftarrow \tau}{\Delta; \Gamma \vdash \Lambda\psi.e \Leftarrow \Pi\psi{:}W.\tau}\ \psi \text{ is new} \qquad \frac{\Delta; \Gamma \vdash e \Rightarrow \Pi\psi{:}W.\tau \quad \vdash \Psi \Leftarrow W}{\Delta; \Gamma \vdash e\ [\Psi] \Rightarrow [\![\Psi/\psi]\!]\tau}$$

$$\frac{\Delta; \Gamma, f{:}\tau \vdash e \Leftarrow \tau}{\Delta; \Gamma \vdash \mathsf{rec}\ f.e \Leftarrow \tau} \qquad \frac{\Delta; \Gamma, y{:}\tau_1 \vdash e \Leftarrow \tau_2}{\Delta; \Gamma \vdash \mathsf{fn}\ y.e \Leftarrow \tau_1 \to \tau_2}$$

$$\frac{\Delta; \Psi \vdash M \Leftarrow A \quad \Psi \leq \Psi'}{\Delta; \Gamma \vdash \mathsf{box}(\Psi.\, M) \Leftarrow A[\Psi']} \qquad \frac{\Delta; \Psi \vdash \sigma \Leftarrow \Phi \quad \Psi \leq \Psi'}{\Delta; \Gamma \vdash \mathsf{sbox}(\Psi.\, \sigma) \Leftarrow \Phi[\Psi']}\ *$$

$$\frac{\Delta; \Gamma \vdash e \Rightarrow \tau_1 \quad \text{for all } i\ \Delta; \cdot; \Gamma \vdash p_i \Leftarrow (\tau_1 \to \tau)}{\Delta; \Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ p_1 \mid \ldots \mid p_n\ \Leftarrow \tau}$$

$$\frac{\Delta; \Gamma \vdash e \Rightarrow \tau' \quad \tau = \tau'}{\Delta; \Gamma \vdash e \Leftarrow \tau} \qquad \frac{\Delta; \Gamma \vdash e \Leftarrow \tau}{\Delta; \Gamma \vdash (e : \tau) \Rightarrow \tau}$$

$$\frac{y{:}\tau \in \Gamma}{\Delta; \Gamma \vdash y \Rightarrow \tau} \qquad \frac{\Delta; \Gamma \vdash e_1 \Rightarrow \tau_2 \to \tau \quad \Delta; \Gamma \vdash e_2 \Leftarrow \tau_2}{\Delta; \Gamma \vdash e_1\, e_2 \Rightarrow \tau}$$

We only point out a few interesting issues. First the typing rule for $\mathsf{box}(\Psi.\, M)$. M denotes a object-level term whose free variables are defined in the context Ψ, i.e. it is closed with respect to a context Ψ. To type $\mathsf{box}(\Psi.\, M)$ we switch to object-level typing, and forget about the previous context Γ which only describes assumptions on the computation-level. Similar reasoning holds for the typing rule for $\mathsf{sbox}(\Psi.\, \sigma)$. To access data, we provide a case-statement with pattern matching. The intention is to match against the contextual modal variables occurring in the pattern.

Branches

$$\frac{\Delta; (\Delta', u{::}A[\Phi]); \Gamma \vdash p \Leftarrow (\tau_1 \to \tau)}{\Delta; \Delta'; \Gamma \vdash \epsilon u{::}A[\Phi].p \Leftarrow (\tau_1 \to \tau)} \qquad \frac{\Delta, (\Delta', s{::}\Psi[\Phi]); \Gamma \vdash p \Leftarrow (\tau_1 \to \tau)}{\Delta; \Delta'; \Gamma \vdash \epsilon s{::}\Psi[\Phi].p \Leftarrow (\tau_1 \to \tau)}$$

$$\frac{\Delta'; \Gamma \vdash \mathsf{box}(\Psi.\, M) \Leftarrow \tau_1 \quad (\Delta, \Delta'); \Gamma \vdash e \Leftarrow \tau}{\Delta; \Delta'; \Gamma \vdash \mathsf{box}(\Psi.\, M) \mapsto e \Leftarrow (\tau_1 \to \tau)}$$

$$\frac{\Delta'; \Gamma \vdash \mathsf{sbox}(\Psi.\, \sigma) \Leftarrow \tau_1 \quad (\Delta, \Delta'); \Gamma \vdash e \Leftarrow \tau}{\Delta; \Delta'; \Gamma \vdash \mathsf{sbox}(\Psi.\, \sigma) \mapsto e \Leftarrow (\tau_1 \to \tau)}$$

Contexts

$$\frac{}{\psi(\omega) \leq \psi} \qquad \frac{\Psi \leq \Psi'}{\Psi, x{:}A \leq \Psi', x{:}A} \qquad \frac{}{\Delta; \Gamma \vdash \cdot \leq \cdot}$$

Here we also observe the usual bound variable renaming conditions. In the function rule we assume that the variable x is new and does not occur already in Γ. Context variables are explicitly quantified and bound by $\Lambda\psi.e$. In particular, the context variable ψ in $\mathsf{box}(\psi.\, M)$ is not bound by box. Recall also, that we do not allow constraints at binding occurrences of context variables in types. As a consequence, we only compare $\psi(\omega)$ with ψ instead of $\psi(\omega')$.

In the rules for explicit substitutions, marked with *, we need to possibly rename the domain of σ. This can always be achieved. Renaming of the domain of a substitution can be done explicitly by σ'/Ψ. Similarly, in the rule for $\mathsf{box}(\Psi.\, M)$ we may need to rename the variables in Ψ' to match the variables in Ψ.

4 Ordinary and contextual substitutions

In this section we define the operations of substitution. There are multiple substitution operations because we have several different kinds of variables. First, we have program variables x for which we can substitute a computation-level expression. Second, we have object-level variables x for which other objects may be substituted. Third, we have contextual modal variables u for which an open object can be substituted. Fourth, we will have contextual substitutions for substitution and context variables.

Ordinary substitution for program and object-level variables The operations are capture-avoiding and defined in a standard manner. Our convention is that substitutions as defined operations on object-level terms and expressions are written in prefix notation $[M/x]N$ for an object-level substitution and $[e/x]e'$ for computation-level substitution. Note that in $[M/x]N$ the bound variable x denotes an object-level term, while in the $[e/x]e'$ the bound variable x denotes a computation-level expression. We only show the substitutions on the computation-level to illustrate some basic principles. The details for $[M/x]N$ can be found in [NPP05].

Substitution for computation-level expressions

$$
\begin{array}{lcll}
[e/x](x) & = & e & \\
[e/x](y) & = & y \quad \text{if } y \neq x & \\
[e/x](\Lambda\psi.e') & = & \Lambda\psi.[e/x]e' & \text{provided } \psi \notin \mathsf{FV}(e) \\
[e/x](e'\ \lceil\Psi\rceil) & = & [e/x]e'\ \lceil\Psi\rceil & \\
[e/x](\mathsf{fn}\ y.e') & = & \mathsf{fn}\ y.[e/x]e' & \text{provided } y \notin \mathsf{FV}(e) \text{ and } y \neq x \\
[e/x](\mathsf{rec}\ f.e') & = & \mathsf{rec}\ f.[e/x]e' & \text{provided } f \notin \mathsf{FV}(e) \text{ and } f \neq x \\
[e/x](e_1\ e_2) & = & ([e/x]e_1)\ ([e/x]e_2) & \\
[e/x](\mathsf{box}(\Psi.\, M)) & = & \mathsf{box}(\Psi.\, M) & \\
[e/x](\mathsf{sbox}(\Psi.\, \sigma)) & = & \mathsf{sbox}(\Psi.\, \sigma) & \\
[e/x](\mathsf{case}\ e'\ \mathsf{of}\ p_1 \mid \ldots \mid p_n\ ) & = & \mathsf{case}\ [e/x]e'\ \mathsf{of}\ [e/x]p_1 \mid \ldots \mid [e/x]p_n &
\end{array}
$$

Substitution for computation-level patterns

$$
\begin{array}{lcl}
[e/x](\epsilon u{::}A[\Phi].p) & = & \epsilon u{::}A[\Phi].[e/x]p \\
[e/x](\epsilon s{::}\Psi[\Phi].p) & = & \epsilon s{::}\Psi[\Phi].[e/x]p \\
[e/x](\mathsf{box}(\Psi.\, M) \mapsto e') & = & \mathsf{box}(\Psi.\, M) \mapsto [e/x]e' \\
[e/x](\mathsf{sbox}(\Psi.\, \sigma) \mapsto e') & = & \mathsf{sbox}(\Psi.\, \sigma) \mapsto [e/x]e'
\end{array}
$$

Note that $\mathsf{box}(\Psi.\, M)$ does not contain any free occurrences of program variables x, and therefore substitution has no effect. Similarly, the case for $\mathsf{sbox}(\Psi.\, \sigma)$ where no change is visible when $[e/x]$ is applied to it.

Theorem 1 (Substitution on computation-level variables)
If $\Delta; \Gamma \vdash e : \tau$ *and* $\Delta; \Gamma, x{:}\tau, \Gamma' \vdash e' : \tau'$ *then* $\Delta; \Gamma, \Gamma' \vdash [e/x]e' : \tau'$.

Proof. By induction on the structure of the second given derivation.

A similar substitution principle holds for object-level variables.

Theorem 2 (Substitution on object-level variables)
If $\Delta; \Psi \vdash M \Leftarrow A$ *and* $\Delta; \Psi, x{:}A, \Psi' \vdash J$ *then* $\Delta; \Psi, \Psi' \vdash [M/x]J$.

Proof. By induction on the structure of the second given derivation.

Contextual substitution for meta-variables Substitutions for contextual variables u are a little more difficult. We can think of $u[\sigma]$ as a closure where as soon as we know which term u should stand for we can apply σ to it. Because of α-conversion, the variables that are substituted at different occurrences of u may be different, contextual substitution for a meta-variable must carry a context, written as $[\![\Psi.M/u]\!]N$, $[\Psi.M/u]\sigma$, and $[\![\Psi.M/u]e$ where Ψ binds all free variables in M. This complication can be eliminated in an implementation of our calculus based on de Bruijn indexes.

Contextual substitution into objects-level terms

$$
\begin{array}{lcl}
[\![\Psi.M/u]\!](x) & = & x \\
[\![\Psi.M/u]\!](\lambda y.N) & = & \lambda y.[\![\Psi.M/u]\!]N \\
[\![\Psi.M/u]\!](R\ N) & = & ([\![\Psi.M/u]\!]R)\ ([\![\Psi.M/u]\!]N) \\
[\![\Psi.M/u]\!](u[\sigma]) & = & [[\![\Psi.M/u]\!]\sigma/\Psi]M \\
[\![\Psi.M/u]\!](v[\sigma]) & = & v[[\![\Psi.M/u]\!]\sigma] \quad \text{provided } v \neq u \\
[\![\Psi.M/u]\!](\cdot) & = & \cdot \\
[\![\Psi.M/u]\!](\sigma, N/y) & = & [\![\Psi.M/u]\!]\sigma, ([\![\Psi.M/u]\!]N)/y \\
[\![\Psi.M/u]\!](s[\rho]) & = & [\![s[([\![\Psi.M/u]\!]\rho)]
\end{array}
$$

Applying $[\![\Psi.M/u]\!]$ to the closure $u[\sigma]$ first obtains the simultaneous substitution $\sigma' = [\![\Psi.M/u]\!]\sigma$, but instead of returning $M[\sigma']$, it proceeds to eagerly apply σ' to M. Before σ' can be carried out, however, it's domain must be renamed to match the variables in Ψ, denoted by σ'/Ψ.

While the definition of the discussed case may seem circular at first, it is actually well-founded. The computation of σ' recursively invokes $[\![\Psi.M/u]\!]$ on σ, a constituent of $u[\sigma]$. Then σ'/Ψ is applied to M, but applying simultaneous substitutions has already been defined without appeal to meta-variable substitution.

Contextual substitution into computation-level expressions

$$
\begin{array}{lcl}
[\![\Psi.M/u]\!](x) & = & x \\
[\![\Psi.M/u]\!](\Lambda\psi.e) & = & \Lambda\psi.[\![\Psi.M/u]\!]e \\
[\![\Psi.M/u]\!](e\ \lceil\Psi\rceil) & = & ([\![\Psi.M/u]\!]e)\ \lceil\Psi\rceil \\
[\![\Psi.M/u]\!](\mathsf{rec}\ f.e) & = & \mathsf{rec}\ f.[\![\Psi.M/u]\!]e \\
[\![\Psi.M/u]\!](\mathsf{fn}\ y.e) & = & \mathsf{fn}\ y.[\![\Psi.M/u]\!]e \\
[\![\Psi.M/u]\!](e_1\ e_2) & = & ([\![\Psi.M/u]\!]e_1)\ ([\![\Psi.M/u]\!]e_2) \\
[\![\Psi.M/u]\!](\mathsf{box}(\Phi.\,N)) & = & \mathsf{box}(\Phi.\,[\![\Psi.M/u]\!]N) \\
[\![\Psi.M/u]\!](\mathsf{sbox}(\Phi.\,\sigma)) & = & \mathsf{sbox}(\Phi.\,[\![\Psi.M/u]\!]\sigma) \\
[\![\Psi.M/u]\!](\mathsf{case}\ e\ \mathsf{of}\ p_1 \mid \ldots \mid p_n\ ) & = & \mathsf{case}\ [\![\Psi.M/u]\!]e_1\ \mathsf{of}\ [\![\Psi.M/u]\!]p_1 \mid \ldots \mid [\![\Psi.M/u]\!]p_n
\end{array}
$$

The cases for function and recursion do not have to consider capture-avoiding side conditions. Since M must be closed with respect to Ψ and meta-variables u are distinct from computation-level variables x no clashes can happen.

Contextual substitution into computation-level branches

$$
\begin{array}{lcl}
[\![\Psi.M/u]\!](\epsilon v{::}B[\Phi].p) & = & \epsilon v{::}B[\Phi].[\![\Psi.M/u]\!]p \quad \text{provided } v \notin \mathsf{FMV}(M) \text{ and } v \neq u \\
[\![\Psi.M/u]\!](\epsilon s{::}\Psi[\Phi].p) & = & \epsilon s{::}\Psi[\Phi].[\![\Psi.M/u]\!]p \quad \text{provided } s \notin \mathsf{FMV}(M) \\
[\![\Psi.M/u]\!](\mathsf{box}(\Phi.\,N) \mapsto e)) & = & \mathsf{box}(\Phi.\,N) \mapsto [\![\Psi.M/u]\!]e \\
[\![\Psi.M/u]\!](\mathsf{sbox}(\Phi.\,\sigma) \mapsto e)) & = & \mathsf{sbox}(\Phi.\,\sigma) \mapsto [\![\Psi.M/u]\!]e
\end{array}
$$

Finally, the cases for the branches are interesting. We require that all the contextual variables occurring in a pattern $N \mapsto e$ are explicitly quantified by ϵ and hence we do not apply the contextual substitution $[\![\Psi.M/u]\!]$ to the object N describing the pattern, but only to e. Contextual substitution satisfy the following substitution property.

Theorem 3

1. *If* $\Delta; \Psi \vdash M : A$ *and* $\Delta, u{::}A[\Psi]; \Gamma \vdash e : \tau$ *then* $\Delta; \Gamma \vdash [\![\Psi.M/u]\!]e : \tau$.
2. *If* $\Delta; \Psi \vdash M : A$ *and* $\Delta, u{::}A[\Psi]; \Phi \vdash J$ *then* $\Delta; \Phi \vdash [\![\Psi.M/u]\!]J$.

Contextual substitution for context variables Next, we consider substitutions for context variables. Unlike the previous substitution operations which were total, substitution of a context Ψ into a context variable $\psi(\omega)$ may fail if Ψ does not satisfy ω.

Context substitution into computation-level expressions

$$
\begin{array}{lcl}
[\![\Psi/\psi]\!](x) & = & x \\
[\![\Psi/\psi]\!](\Lambda\gamma.e) & = & \Lambda\gamma.[\![\Psi/\psi]\!]e \quad \text{provided that } \gamma \notin \mathsf{FV}(\Psi) \\
[\![\Psi/\psi]\!](e\ \lceil\Phi\rceil) & = & ([\![\Psi/\psi]\!]e)\ \lceil[\![\Psi/\psi]\!]\Phi\rceil \\
[\![\Psi/\psi]\!](\mathsf{fn}\ y.e) & = & \mathsf{fn}\ y.[\![\Psi/\psi]\!]e \\
[\![\Psi/\psi]\!](\mathsf{rec}\ f.e) & = & \mathsf{rec}\ f.[\![\Psi/\psi]\!]e \\
[\![\Psi/\psi]\!](e_1\ e_2) & = & ([\![\Psi/\psi]\!]e_1)\ ([\![\Psi/\psi]\!]e_2) \\
[\![\Psi/\psi]\!](\mathsf{box}(\Phi.N)) & = & \mathsf{box}([\![\Psi/\psi]\!]\Phi.\ [\![\Psi/\psi]\!]N) \\
[\![\Psi/\psi]\!](\mathsf{sbox}(\Phi.\sigma)) & = & \mathsf{sbox}(\Phi.\ [\![\Psi/\psi]\!]\sigma) \\
[\![\Psi/\psi]\!](\mathsf{case}\ e\ \mathsf{of}\ p_1 \mid \ldots \mid p_n\ ) & = & \mathsf{case}\ [\![\Psi/\psi]\!]e_1\ \mathsf{of}\ q_1 \mid \ldots \mid q_k \\
 & & \text{where } [\![\Psi/\psi]\!]p_i = q_i \text{ for some } i
\end{array}
$$

In the case for $\mathsf{box}(\Phi.N)$ we apply the substitution $[\![\Psi/\psi]\!]$ to both the context Φ and the object N. While the object N does not contain context variables ψ, it may contain the identity substitution $\mathsf{id}_(\psi)$ which needs to be unfolded. Similar considerations hold for the case $\mathsf{sbox}(\Phi.\sigma)$. Note that applying substitution $[\![\Psi/\psi]\!]$ to some of the branches p_i may actually fail, and the substitution operation eliminates these branches, since they are unreachable. Coverage checking [SP03] will guarantee that there is at least one branch where applying substitution $[\![\Psi/\psi]\!]$ to it will succeed. Hence applying the substitution $[\![\Psi/\psi]\!]$ to an expression is only total if we covered all possible cases which guarantees that there must be at least one case where applying $[\![\Psi/\psi]\!]$ to the branch succeeds. The most interesting case is where actual substitution must happen.

$$
\begin{array}{lcll}
[\![\Psi/\psi]\!](\cdot) & = & \cdot & \\
[\![\Psi/\psi]\!](\Phi, x{:}A) & = & ([\![\Psi/\psi]\!]\Phi), x{:}A & \text{provided } x \notin \mathsf{V}([\![\Psi/\psi]\!]\Phi) \\
[\![\Psi/\psi]\!](\psi(\omega)) & = & \Psi & \text{if } \Psi \text{ satisfies } \omega. \\
[\![\Psi/\psi]\!](\phi(\omega)) & = & \phi & \text{for } \phi \neq \psi.
\end{array}
$$

$$
\begin{array}{lcll}
[\![\Psi/\psi]\!](\cdot) & = & \cdot & \\
[\![\Psi/\psi]\!](\sigma, M/x) & = & [\![\Psi/\psi]\!]\sigma, [\![\Psi/\psi]\!]M/x & \\
[\![\Psi/\psi]\!](s[\rho]) & = & s[[\![\Psi/\psi]\!]\rho] & \\
[\![\Psi/\psi]\!](\mathsf{id}_{\psi(\omega)}) & = & \mathsf{id}(\Psi) & \text{if } \Psi \text{ satisfies } \omega. \\
[\![\Psi/\psi]\!](\mathsf{id}_{\phi(\omega)}) & = & \mathsf{id}_{\phi(\omega)} &
\end{array}
$$

We recall again that no construct binds a context variable ψ. In particular, $\mathsf{box}(\psi.M)$ does not bind ψ. Expansion of the identity substitution is defined as follows:

$$
\begin{array}{lcl}
\mathsf{id}(\cdot) & = & \cdot \\
\mathsf{id}(\Psi, x{:}A) & = & \mathsf{id}(\Psi), x/x \\
\mathsf{id}(\psi(\omega)) & = & \mathsf{id}_{\psi(\omega)}
\end{array}
$$

Lemma 1 *If* $\mathsf{id}(\Psi) = \sigma$ *then* $\Delta; \Psi, \Psi' \vdash \sigma \Leftarrow \Psi$.

Lemma 2 *If* $\Delta, \psi{:}W; \psi(\omega), \Phi \vdash M \Leftarrow A$ *and* $\Psi : W$ *and* Ψ *satisfies* ω *then* $\Delta; \Psi, \Phi \vdash M \Leftarrow A$.

Next, we give a brief definition for substituting for substitution variables.

Substitution into objects-level terms

$$
\begin{array}{lcl}
[\![\Psi.\sigma/s]\!](x) & = & x \\
[\![\Psi.\sigma/s]\!](\lambda y N) & = & \lambda y [\![\Psi.\sigma/s]\!]N \\
[\![\Psi.\sigma/s]\!](N_1\ N_2) & = & ([\![\Psi.\sigma/s]\!]N_1)\ ([\![\Psi.\sigma/s]\!]N_2) \\
[\![\Psi.\sigma/s]\!](u[\rho]) & = & u[[\![\Psi.\sigma/s]\!]\rho] \\
[\![\Psi.\sigma/s]\!](\cdot) & = & \cdot \\
[\![\Psi.\sigma/s]\!](\sigma, N/y) & = & [\![\Psi.\sigma/s]\!]\sigma, ([\![\Psi.\sigma/s]\!]N)/y \\
[\![\Psi.\sigma/s]\!](s[\rho]) & = & [\![[([\![\Psi.\sigma/s]\!]\rho)/\Psi]\sigma \\
[\![\Psi.\sigma/s]\!](s'[\rho]) & = & [\![s'[([\![\Psi.\sigma/s]\!]\rho)]
\end{array}
$$

Applying $[\![\Psi.\sigma/s]\!]$ to the closure $s[\rho]$ first obtains the simultaneous substitution $\rho' = [\![\Psi.\sigma/s]\!]\rho$, but instead of returning $sigma[\rho']$, it proceeds to eagerly apply ρ' to σ. Before ρ' can be carried out, however, it's domain must be renamed to match the variables in Ψ, denoted by ρ'/Ψ.

Lemma 3 *If* $\Delta; \Psi \vdash \sigma \Leftarrow \Psi'$ *and* $\Delta, s{::}\Psi'[\Psi]; \Phi \vdash M \Leftarrow A$
then $\Delta; \Phi \vdash [\![\Psi.\sigma/s]\!]J$.

Proof. By structural induction on the second derivation.

5 Operational semantics

In this section, we sketch a small-step operational semantics for this language. First, we define the values in this language.

$$\text{Value} \quad v \quad ::= \quad \mathsf{fn}\ y.e \mid \mathsf{box}(\Psi.\, M) \mid \mathsf{sbox}(\Psi.\, \sigma)$$

Next, we define a small-step evaluation judgment:

$$
\begin{array}{lll}
 & e \longrightarrow e' & \text{Expression } e \text{ evaluates in one step to } e'. \\
\Delta \vdash & \mathsf{box}(\Psi.\, M) \doteq p \Rightarrow e' & \text{Branch } p \text{ matches } \mathsf{box}(\Psi.\, M) \text{ and steps to } e' \\
\Delta \vdash & \mathsf{sbox}(\Psi.\, \sigma) \doteq p \Rightarrow e' & \text{Branch } p \text{ matches to } \mathsf{sbox}(\Psi.\, \sigma) \text{ and steps to } e'
\end{array}
$$

We only concentrate on three interesting cases, where actual computation happens. The case for function application is straightforward. Values for program variables are propagated by computation-level substitution. Next the case for pattern matching against $\mathsf{box}(\Psi.\, M)$ and $\mathsf{sbox}(\Psi.\, \sigma)$. Here we need to propagate object-level terms via contextual substitution.

$$\frac{}{(\mathsf{fn}\ y{:}\tau.e)\ v \longrightarrow [v/y]e}$$

$$\frac{\cdot \vdash \mathsf{box}(\Psi.\, M) \doteq p_i \Rightarrow e'}{(\mathsf{case}\ (\mathsf{box}(\Psi.\, M))\ \mathsf{of}\ p_1 \mid \ldots \mid p_n\ ) \longrightarrow e'}$$

$$\frac{\cdot \vdash \mathsf{sbox}(\Psi.\, \sigma) \doteq p_i \Rightarrow e'}{(\mathsf{case}\ (\mathsf{sbox}(\Psi.\, \sigma))\ \mathsf{of}\ p_1 \mid \ldots \mid p_n\ ) \longrightarrow e'}$$

Since evaluation relies on pattern matching object-level terms, we describe briefly this process. In particular, we rely on higher-order pattern matching to match $\mathsf{box}(\Psi.\, M)$ against $\mathsf{box}(\Psi.\, M') \mapsto e$. Higher-order patterns in the sense of Miller [Mil91] restrict syntactically the occurrences of contextual modal variables $u[\sigma]$. The pattern restriction enforces that the substitution σ which is associated with the contextual modal variable u only maps variables to variables and has the following form: $y_1/x_1, \ldots, y_n/x_n$. This ensures

that higher-order pattern matching remains decidable in the presence of λ-abstraction. The judgment for higher-order pattern matching can be described as follows:

$$\Delta; \Psi \vdash M \doteq M'/\theta \quad M \text{ matches } M' \text{ s.t. } [\![\theta]\!]M' = M$$

A description of higher-order pattern matching for contextual modal variables can be found in [Pie03]. It seems feasible to extended this description to incorporate also substitution while preserving correctness and crucial invariants of higher-order pattern matching such as

1. θ has domain Δ and instantiates all modal variables in Δ.
2. $M = [\![\theta]\!]N$, i.e. object M is syntactically equal to $[\![\theta]\!]N$.

We are now in a position to describe computation-level pattern matching of $\mathsf{box}(\Psi.\, M)$ against a pattern p. Pattern matching of $\mathsf{sbox}(\Psi.\, \sigma)$ follows similar ideas.

$$\frac{\Delta, v{::}A[\Phi] \vdash \mathsf{box}(\Psi.\, M) \doteq p \Rightarrow e'}{\Delta \vdash \mathsf{box}(\Psi.\, M) \doteq (\epsilon v{::}A[\Phi].p) \Rightarrow e'}$$

$$\frac{\Delta, s{::}\Phi'[\Phi] \vdash \mathsf{box}(\Psi.\, M) \doteq p \Rightarrow e'}{\Delta \vdash \mathsf{box}(\Psi.\, M) \doteq (\epsilon s{::}\Phi'[\Phi].p) \Rightarrow e'}$$

$$\frac{\Delta; \Psi \vdash M \doteq M'/\theta}{\Delta \vdash \mathsf{box}(\Psi.\, M) \doteq (\mathsf{box}(\Psi.\, M') \mapsto e) \Rightarrow [\![\theta]\!]e}$$

In the future, we plan to extend this foundation to incorporate matching against context variables. Given the current setup, we conjecture type safety for our proposed functional language with higher-order abstract syntax and explicit substitutions.

6 Examples

In this section, we will show several examples to illustrate the potential applications of the ideas presented. All the examples require context variables to denote an open world we recurse over. This world is known during run-time, but changes. The examples also make essential use of substitution variables and first-class substitutions. These features can be used to model first-class environments. For some of the examples in this section, the given foundation is not yet strong enough, but matching on context variables is required. Nevertheless we include these examples to illustrate our long term vision.

Variable counting First, we show a very simple function which counts the bound variables occurring in an expression defined using higher-order abstract syntax. We assume, we have declared a data-type exp for expressions using higher-order abstract syntax which contains the objects $\mathsf{lam} : (\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp}$ and $\mathsf{app} : \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp}$.. We assume we have available basic computation-level types such as nat, string or bool.

$$\begin{array}{l}
\mathsf{rec}\ cnt. \\
\quad \Lambda\gamma.\mathsf{fn}\ e.\mathsf{case}\ e\ \mathsf{of}\ \mathsf{box}(\gamma(x{:}exp).\, x) \qquad\qquad\qquad\qquad \mapsto 1 \\
\quad\quad \mid \epsilon u{::}\mathsf{exp}[\gamma, x'{:}\mathsf{exp}]. \quad \mathsf{box}(\gamma.\, \mathsf{lam}\ \lambda x{:}\mathsf{exp}.u[\mathsf{id}_\gamma, x/x']) \quad \mapsto \\
\qquad\qquad\qquad\qquad\qquad\qquad cnt\ [\gamma, x{:}\mathsf{exp}]\ (\mathsf{box}(\gamma, x{:}\mathsf{exp}.\, u[\mathsf{id}_\gamma, x/x'])) \\
\quad\quad \mid \epsilon u{::}\mathsf{exp}[\gamma]\ \epsilon v{::}\mathsf{exp}[\gamma]. \quad \mathsf{box}(\gamma.\, \mathsf{app}\ u[\mathsf{id}_\gamma]\ v[\mathsf{id}_\gamma]) \quad \mapsto \\
\qquad\qquad\qquad\qquad\qquad\qquad cnt\ [\gamma]\ (\mathsf{box}(\gamma.\, u[\mathsf{id}_\gamma])) + cnt\ [\gamma]\ (\mathsf{box}(\gamma.\, v[\mathsf{id}_\gamma]))
\end{array}$$

Note we need to use a context variable γ to denote our context of variables which may occur in the term e and which will be built up during recursion. Only during runtime, do we know the actual context. Omitting some type information (which we think of being implicit) and following Twelf-like syntax where λ-abstraction is denoted by `[x] ...` this can be beautified to:

```
rec cnt: Pi g:expW. exp[g] -> int =
Lam g => fn e =>
case e of
  (box g(x). x)
     => 1
 |(box g. (app u[id_g] v[id_g]))
     => cnt [g] (box g. u[id_g]) +  cnt(box g. v[id_g])
 |(box g. (lam [x] u[id_g, x/x])
     => cnt [g,x] (box g,x. u[id_g, x/x])
```

Substitution-based and environment-based evaluator Assume we have the following data-type declaration for numbers and expressions. We use higher-order abstract syntax to denote the binder in the let-expression.

```
z nat.
suc: nat -> nat.
num: nat -> exp
Add: exp -> exp -> exp.
Let: exp -> (nat -> exp) -> exp.
```

We assume we defined a function for addition `add:nat[]*nat[] -> nat[]`. Then we can define a simple evaluator in a straightforward way. If we encounter a let-expression `let x = e in e' end` then we first evaluate the expression `e` to some value `v`, and then we replace all occurrences of the binder `x` in `e'` with the value `v`. This is handled by building the closure `u[id_g,v[]/x]`. It should be obvious how to extend this evaluator to other forms of arithmetic expressions.

```
rec eval: exp[] ->  nat[] =
fn e : exp[] =>
case e of
  (box . Nat n[])  => (box . n[])
| (box . Add(e1[], e2[])) =>
  let
    val a = eval (box . e1[])
    val b = eval (box . e2[])
  in
    add (a, b)
  end
| (box . Let(e[], [x]. u[x/x]))  =>
  let
    box v = eval (box . e[])
   in
    eval(box . u[v[]/x])
  end
```

While the substitution-model has many advantages from a theoretical point of view, in an implementation it is usually considered too expensive. Alternatively, we can use an environment model where we associates variables with values in an environment. When we evaluate a let-expression `let x = e in e' end` then we evaluate the expression `e` to some value `v`, and then evaluate `e'` in an environment where we associate the binder `x` with the value `v`. When we encounter a variable `x`, we lookup its value in the environment. We will use explicit substitutions to model the run-time environment. To ease readability, we pattern match simultaneously on both inputs and include pattern matching on context variables. `[]` denotes the empty context in `g[]`, and the empty substitution in `s[]` or `v[]`.

```
natW = x_1:nat, ... x_n:nat
(* eval: Pi g:natW. exp[g] -> g[] -> nat[] *)
fun eval [g,x]    (box g,x. x) (sbox s[], v[]/x) =
      box(. v[])
  | eval [g(x),y] (box g(x),y. x) (sbox s[], v[]/y) =
      eval [g(x),y] (box g(x).x) (sbox s[])
  | eval [g] (box g. Nat n[]) (env) = (box . n[])
  | eval [g] (box g. Add(e1[id_g], e2[id_g])) env =
  let
    val a = eval [g] (box g. e1[id_g]) env
    val b = eval [g] (box g. e2[id_g]) env
  in
    add(a, b)
  end
  | eval [g] (box g. Let(e, [x]. u[id_g, x/x])) (sbox s[]) =
  let
    val box v = eval [g] (box g. e[id_g]) (sbox s[])
  in
    eval [g,x]  (box g,x. u[id_g,x/x]) (sbox s[], v[]/x)
  end
```

Since `box g.e` guarantees that all the "free" variables occurring in the expression `e` are bound in the context `g`. Since we can have terms together with their local context such as `box g.e`, and we allow pattern matching against local contexts, we can also directly write the case for variables. The type system also guarantees that the environment `sbox s[]` provides instantiations for every variable in the local context `g`. Finally, the type system ensures that the instantiations for each of the variables in the local context are closed. The type system therefore allows us to type-check dependencies between substitutions and open terms and enforces that they can be linked safely.

7 Related Work

Techniques for supporting higher-order abstract encodings in functional programming languages have received wide spread attention. One of the first proposals for functional programming with support for binders and higher-order abstract syntax was presented by Miller [Mil90]. Later, Despeyroux, Pfenning, and Schürmann, have developed proof-theoretic foundations for programming and reasoning with higher-order abstract syntax [DPS97, SPS05] based on modal types. However, there are no contextual types and their theoretical development lacks first-class meta-variables and a context of meta-variables. This has deep consequences for the theoretical development. Since in our framework the type of a meta-variable determines its local scope, local scope is naturally enforced. No scope stacks and operations on them is required, which should simplify the theoretical development. For example, the operational semantics does not have to take into account a stack of contexts and there are no explicit operations for popping a context of the stack in the proposed language. In addition, context abstraction in our setting allows us to enforce stronger invariants about programs since we can distinguish between different context and different worlds. The nature of the nabla-quantifier allows only reasoning within one world or context. Moreover, we propose to extend this framework with explicit substitutions, which are interesting in its own.

Westerbrook [Wes06] presents a preliminary proposal for programming with higher-order abstract syntax with dependent records. Similar to Schürmann's work [SPS05] fresh variables can be introduced via the nabla-quantifier. To describe open terms, i.e. terms which contain free variables, he relies on modalities which are indexed by variable lists. While the paper claims to draw on previous idea in [NPP05], it is very different in its flavour. For example, in [NPP05] and in this paper, we consider objects of type $A[\Psi]$ to be closed with respect to Ψ. This is evident in the typing rule where we switch context and in our definition of

contextual substitution. Another important feature of our work is alpha-conversion. We consider the types $A[x : B]$ to be alpha-equivalent to $A[y : B]$. Finally, none of the discussed approaches allows computation with open terms. Open terms may arise during computation, but they lack a first-class status.

In functional programming, various formulations of context as a primitive programming construct have been considered [SSB01, SSK02, Nis00, Mas99, HO01]. Nishizaki [Nis00] for example extends a lambda-calculus with explicit substitutions in the spirit of the explicit substitution calculus proposed by Abadi *et. al.*[ACCL90]. However, unlike Abadi's work, the author proposes a polymorphic calculus where we can quantify over explicit substitutions. This work crucially relies on de Bruijn indices. Although the use of de Bruijn indices is useful in an implementation, nameless representation of variables via de Bruijn indices are usually hard to read and critical principles are obfuscated by the technical notation M. Sato et al. [SSK02] introduce a simply typed λ-calculus which has both contexts and environments (= substitutions) as first-class values, called $\lambda_{\kappa,\epsilon}$-calculus. There are many distinctions, however, between $\lambda\kappa\epsilon$ and the contextual modal type theory we propose as a foundation. Most of these differences arise because the former is not based on modal logic. For example, they do not allow α-conversion open objects and they do not require open objects to be well-formed with respect to a local context. Moreover, they do not cleanly distinguish between meta-variables and ordinary variables. All these restrictions together make their system quite heavy, and requires fancy substitution operations and levels attached to ordinary variables to maintain decidability and confluence.

8 Conclusion future plans

We have sketched a foundation for functional programming with higher-order abstract syntax and explicit substitution based on contextual modal types. Our proposal builds on earlier ideas by [DPS97, SPS05] where modal types have been used to distinguish between object-level terms and computation-level programs. In contrast to earlier proposals, we distinguish between contextual meta-variables and ordinary variables which we believe may lead to a cleaner and more expressive framework for programming with higher-order abstract syntax. The distinction between contextual meta-variables and ordinary bound variables has already provided interesting insights into higher-order proof search, higher-order unification and logical frameworks in general (see for example [PP03, Pie03]). It has allowed us to clarify many theoretical issues and invariants related to the interplay of meta-variables (= contextual variables) and ordinary variables. Contextual modal types also have been applied to staged functional programming to generate code which may possibly be open [NPP05]. In this paper, we apply contextual modal types to programming with open terms based on higher-order abstract syntax.

There are many aspects left to consider. In particular, the status of context variables and the operations allowed on them needs to be expanded for many practical examples. Another important aspect of this work is that our proposal ensures the adequate encoding of on-paper formulations. We believe that it is possible to prove adequacy about our examples since our object-level theory draws on ideas from logical frameworks, and adequacy proofs for encodings within logical frameworks are standard. More interestingly we would like to consider adding datatypes defined via higher-order abstract syntax with ordinary types. A more practical aspect is type-inference and how much typing information can be omitted in practice. Currently we are working on a prototype which will provide valuable insights into many of the practical issues arising and will allow us to experiment with a wide variety of sample programs from different areas.

References

[ACCL90] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. Explicit substitutions. In *Symposium on Principles of Programming Languages, POPL'90*, pages 31–46, San Francisco, California, 1990. ACM.

[DPS97] Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In R. Hindley, editor, *International Conference on Typed Lambda Calculus and*

Applications, TLCA'97, pages 147–163, Nancy, France, April 1997. Springer-Verlag LNCS. An extended version is available as Technical Report CMU-CS-96-172, Carnegie Mellon University.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[HO01] Masatomo Hashimoto and Atsushi Ohori. A typed context calculus. *Theoretical Computer Science*, 266(1–2):249–272, 2001.

[Mas99] Ian A. Mason. Computing with contexts. *Higher-Order and Symbolic Computation*, 12(2):171–201, 1999.

[Mil90] Dale Miller. An extension to ml to handle bound variables in data structures. In G. Huet and G. Plotkin, editors, *Proceedings of the First Workshop on Logical Frameworks*, pages 323–335, 1990.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[Nis00] Shin-Ya Nishizaki. A polymorphic environment calculus and its type-inference algorithm. *Higher Order Symbol. Comput.*, 13(3):239–278, 2000.

[NPP05] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *submitted*, 2005.

[Pie03] Brigitte Pientka. *Tabled higher-order logic programming.* PhD thesis, Computer Science Department, Carnegie Mellon University, December 2003.

[PP03] Brigitte Pientka and Frank Pfennning. Optimizing higher-order pattern unification. In F. Baader, editor, *International Conference on Automated Deduction, CADE'03*, Lecture Notes in Computer Science (LNAI 2741), pages 473–487, Miami, Florida, 2003. Springer.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In H. Ganzinger, editor, *International Conference on Automated Deduction, CADE'99*, volume 1632 of *Lecture Notes in Artificial Inteligence*, pages 202–206, Trento, Italy, 1999. Springer-Verlag.

[SP03] Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, pages 120–135, Rome, Italy, September 2003. Springer-Verlag LNCS 2758.

[SPS05] Carsten Schürmann, Adam Poswolsky, and Jeffrey Sarnat. The ∇-calculus. functional programming with higher-order encodings. In Pawel Urzyczyn, editor, *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications(TLCA'05), Nara, Japan, April 21-23, 2005*, volume 3461 of *Lecture Notes in Computer Science*, pages 339–353. Springer, 2005.

[SSB01] Masahiko Sato, Takafumi Sakurai, and Rod Burstall. Explicit environments. *Fundamenta Informaticae*, 45(1-2):79–115, 2001.

[SSK02] Masahiko Sato, Takafumi Sakurai, and Yukiyoshi Kameyama. A simply typed context calculus with first-class environments. *Journal of Functional and Logic Programming*, 2002(4), March 2002.

[Wes06] Edwin Westerbrook. Free variable types. Presentation at the Seventh Symposium on Trends in Functional Programming, April 2006.

Position Paper: Towards an Alternation Calculus

Aaron Stump
Computer Science and Engineering
Washington University in St. Louis
St. Louis, Missouri, USA

Abstract

This short paper describes work in progress on a program logic called the Alternation Calculus. This system extends an untyped λ-calculus with an idealized operator which dualizes termination and non-termination. The alternation $\sim M$ converges with a non-deterministically chosen value if all (non-deterministic) computations of M diverge, and diverges if M has a converging computation. Standard logical operators and type operators can be defined as terms using alternation.

1 Introduction

Program verification environments like Coq enable specifications and implementations to be written in the same pure functional programming language [4]. This uniformity is desirable, since it reduces the conceptual burden on developers: they need learn only one language for implementation and specification both, rather than separate languages for each. The main drawback of this approach as realized in Coq is the restriction of the programming language to terminating programs only. From a theoretical and occasionally a practical point of view, requiring termination is a limitation. Furthermore, properties may often be conveniently specified by stipulating that certain possibly non-terminating computations do, in fact, terminate. For example (in an imperative setting), to specify that a structure implemented with pointers in memory is a linked (acyclic) list, we may simply state that a function which follows non-null pointers terminates when called on the first element of the list. This approach can, of course, be simulated using a similar but always terminating function which additionally takes in a counter for how many pointers to follow. But it is arguably cleaner to work with the first, possibly non-terminating function. In that case, the natural proof technique is reasoning by induction on the structure of terminating computations. One may also consider proving relative non-termination by coinduction on the structure of non-terminating computations. Supporting this kind of verification, with general-purpose programs as implementations and specifications and proof by (co-)induction on the structure of computations, is the first goal of this work.

A second goal is inspired by the role of types in Coq. Coq enforces program termination using a combination of typing and other syntactic restrictions. In programming languages without support for full program verification, types play a useful role as automatically verifiable specifications. But once we have chosen to enter the arduous world of program verification, types should be unnecessary, at least in theory, in implementations and specifications. (Programs intended as proofs via the Curry-Howard isomorphism may still need types.) We might expect that automatically checkable types could aid more general program verification. But from a foundational point of view, it would be better if they were redundant, as they are not in Coq, in the implementation language. So, the second goal of the present work is to support verification with untyped programs (as specifications and implementations).

This paper briefly describes initial steps toward a new program logic called the Alternation Calculus (AC) that is intended to support the above goals of verification with untyped, general-purpose programs as implementations and specifications. The Alternation Calculus extends an untyped lambda calculus with an idealized alternation operator, which dualizes termination and non-termination. The alternation $\sim M$ converges with a non-deterministically chosen value if all (non-deterministic) computations of M fail, and

diverges if M has a converging computation. This paper defines the intended evaluation relation of AC, and shows how standard logical operators and type operators can be defined as terms using alternation. Formulating principles of induction and co-induction on the structure of the computation remains to future work. Thus, the paper does not present the Alternation Calculus itself, since no calculus is given for deriving judgements of the declaratively defined evaluation relation. Developing such a calculus remains to future work. Formalization of the meta-theory is ongoing in Coq. The position of this paper is that the above goals are good ones for a program logic, and that the Alternation Calculus is a promising means for achieving them.

There is one existing program logic which already satisfies the above goals. In Dynamic Logic (DL), general programs in an untyped imperative language may be used as both specifications and implementations [2]. Using general programs as specifications does not appear to have been much explored in practical verification with DL. Furthermore, the programming language part of DL is typically taken to be a WHILE-style language. Proof terms are therefore not at all similar to implementations or specifications.

2 Definition of AC

The terms of AC are defined in Figure 1. Informally, we have variables x; applications; call-by-value abstractions (λ); equality ($=$); and alternation ($\tilde{}$). The alternation operator is the central feature of AC. It is similar in spirit to an alternation operator proposed by Nelson, though here we dualize not players in a game, but termination and non-termination [3]. Alternation is called an *idealized* construct (since it is not implementable in general due to the undecidability of halting). The other constructs are called *pure*, and terms built only with them are called pure terms.

To obtain a predicative definition of evaluation, we use a technique proposed recently by others, and define an ω-indexed family $\downarrow_n$ of approximate evaluation relations [1]. Each member of this family cuts off evaluation by returning a special value $\bot$ if the computation tree reaches depth n. Evaluation is defined in Figure 2. A few notes are required. For technical reasons, the rules must also track, in a superscript to $\downarrow$, the maximum number of nested alternations on any path in the computation tree. Also, we work up to α-equivalence of terms. We stipulate that $\bot$ propagates aggressively through the evaluation rules, except where explicitly considered. If a premise has a result which is $\bot$ that is not required to be so or not be so by the rule, then (1) other premises mentioning that result need not be proved and (2) the result in the conclusion of the rule is $\bot$. We use meta-variables (with decorations) T and R for terms and as well $\bot$. We use x for variables. In the premise of F-AltF, we write $T \Downarrow_n^k \bot$ for

$$T \downarrow_n^k \bot \quad \text{and} \quad \forall R.\, \forall k' \leq k.\, (T \downarrow_n^{k'} R \text{ implies } R = \bot)$$

In rule F-Eq, we write $\equiv$ for α-equivalence of terms. Finally, the set $\mathcal{V}$ is the set of pure lambda values, which are λ-abstractions containing no use of alternation. We are interested in the limit of our family of evaluation relations, defined as follows:

$$T \downarrow^k R \quad \Leftrightarrow \quad \exists n.\, \forall n' \geq n.\, \exists k' \leq k.\, T \downarrow_{n'}^{k'} R$$

This says that terms evaluate in the limit if there is a point at which evaluation stabilizes with a bounded number of nested alternations (we often elide the superscript for this number). Nested alternations must be bounded in order to ensure the meta-theoretic property of inversion of the evaluation relation, which is crucial for deriving logical rules. Difficulties with unbounded nested alternation arise due to examples like

$$T ::= x \;\mid\; T_1\ T_2 \;\mid\; \lambda x.T \;\mid\; T_1 = T_2 \;\mid\; \sim T$$

Figure 1: Syntax of AC terms.

the following:

$$\begin{array}{rcl} \mathit{fix} & := & \lambda f.(\lambda x.f\ (\lambda y.x\ x\ y))\ (\lambda x.f\ (\lambda y.x\ x\ y)) \\ \mathit{bad} & := & \mathit{fix}\ (\lambda \mathit{bad}.\lambda x. \sim \mathit{bad}\ x) \end{array}$$

First is a standard call-by-value fixed point combinator. Second is a term which, when applied to an argument, oscillates back and forth between termination and non-termination. This does not evaluate to anything in our limit evaluation relation. In the absence of unbounded nested alternation, limit evaluation in AC is designed to be invertible, though a formal proof of this is not complete yet.

3 Logic

Using alternation, we can define standard logical operators. We take a term T to hold as a formula iff $T \downarrow R$ for $R \neq \bot$ (and write $\star$ for such R). The role of negation is played by alternation. We can then define conjunction, implication, and disjunction like this (where x is not free in T_1):

$$\begin{array}{rcl} T_1\ \wedge\ T_2 & := & (\lambda x.T_1)\ T_2 \\ T_1\ \rightarrow\ T_2 & := & \sim((\lambda x. \sim T_2)\ T_1) \\ T_1\ \vee\ T_2 & := & \sim((\lambda x. \sim T_2)\ \sim T_1) \end{array}$$

The reader may easily confirm that introduction rules for conjunction and disjunction are derivable. Elimination rules then follow by inversion. Implication elimination also clearly holds, by inversion. Implication introduction, on the other hand, holds only with a qualification. Let us call a term T *complete* iff $T \vee \sim T \downarrow \star$. Note that every term in $\mathcal{V}$ is complete. We know from the example term *bad* above that there are incomplete terms.

Theorem 1 *If P is complete, and if $P \downarrow \star$ implies $Q \downarrow \star$, then $P \rightarrow Q \downarrow \star$.*

Proof. Since P is complete, we have $P \vee \sim P \downarrow \star$. By or-elimination, we may reason by cases. Suppose we have $P \downarrow \star$. Then we obtain Q by implication elimination using our hypothesis. Suppose we have $\sim P \downarrow \star$. Then by inversion we have $P \Downarrow \bot$, so the term $(\lambda x. \sim Q)\ P$ also evaluates (with $\Downarrow$) to $\bot$, and the whole implication then evaluates to $\star$. □
We construct the quantifiers in the following way.

$$\begin{array}{rcl} \mathit{fail} & := & \mathit{true} = \mathit{false} \\ \epsilon & := & \sim \mathit{fail} \\ \exists x.\ T & := & (\lambda x.T)\ \epsilon \\ \forall x.\ T & := & \sim \exists x.\ \sim T \end{array}$$

The derivations in Figure 3 show show that ϵ evaluates to any value or to all values, depending on whether we are using $\downarrow$ or $\Downarrow$. Then we get infinitary introduction rules for the quantifiers. Elimination rules again follow by inversion.

4 Simple Types and Church Numerals

This Section shows how simple types can be implemented as unary predicates in AC. We consider simple types formed from a base type *nat* of modified Church numerals, and a simple function space constructor. Consider first these definitions:

$$
\begin{array}{rcl}
\mathit{zero} & := & \lambda s.\lambda z.z \\
\mathit{succ} & := & \lambda n.\lambda s.\lambda z.s\ n \\
\mathit{nat} & := & \mathit{fix}\ (\lambda \mathit{nat}.\lambda x.x = \mathit{zero}\ \vee\ \exists y.\ x = \lambda s.\lambda z.s\ y\ \wedge\ \mathit{nat}\ y)
\end{array}
$$

The definitions for *zero* and *succ* are modified from standard definitions for Church numerals, to enable simpler analysis of terms: to get the predecessor of a numeral, one need just apply it to $\lambda x.x$ (for s) and *zero* (for z), as the reader can confirm. Our base type *nat* is a predicate on AC terms x, which succeeds if either x is zero or equal to the successor of some natural number (non-deterministically guessed). We define the function space constructor as the following AC term.

$$
\Rightarrow\ :=\ \lambda A.\lambda B.\lambda f.\forall x.\ A\ x\ \rightarrow\ f\ x\ \rightarrow\ B\ (f\ x)
$$

This takes in two AC terms A and B, intended as the domain and range types of the function. So then the type $A \Rightarrow B$ is a predicate on terms f. That predicate may be read as saying, "for every value x, if $A\ x$ holds (i.e., has a successful computation), and if $f\ x$ succeeds, then $B\ (f\ x)$ holds." The intention here is to classify just pure terms, which are deterministic. So the reading of $f\ x$ as "$f\ x$ succeeds" as opposed to "$f\ x$ has a successful computation" is justified.

References

[1] E. Ernst, K. Ostermann, and W. Cook. A Virtual Class Calculus. In *The 33rd ACM Symposium on Principles of Programming Languages*, pages 270–282. ACM Press, 2006.

[2] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[3] G. Nelson. Some generalizations and applications of Dijkstra's guarded commands. In M. Broy, editor, *Programming and Mathematical Method*, volume 88 of *NATO ASI Series. Series F: Computer and Systesms Sciences*. Springer-Verlag, 1990.

[4] The Coq Development Team. The Coq Proof Assistant Reference Manual, Version V8.0, 2004. http://coq.inria.fr.

$$\frac{}{\lambda x.T \downarrow^{0}_{n+1} \lambda x.T}\ \text{E-Lam} \qquad \frac{T_1 \downarrow^{k_1}_{n} \lambda x.R_1 \quad T_2 \downarrow^{k_2}_{n} R_2 \quad [R_2/x]R_1 \downarrow^{k_3}_{n} R}{T_1\ T_2 \downarrow^{max(k_1,k_2,k_3)}_{n+1} R}\ \text{E-AppLam}$$

$$\frac{T_1 \downarrow^{k_1}_{n} R \quad T_2 \downarrow^{k_2}_{n} R}{T_1\ =\ T_2 \downarrow^{max(k_1,k_2)}_{n+1} R}\ \text{E-Eq} \qquad \frac{T_1 \downarrow^{k_1}_{n} R \quad T_2 \downarrow^{k_2}_{n} R' \quad R \not\equiv R'}{T_1\ =\ T_2 \downarrow^{max(k_1,k_2)}_{n+1} \bot}\ \text{F-Eq}$$

$$\frac{T \downarrow^{k}_{n} R \quad R \neq \bot}{\sim T \downarrow^{k+1}_{n+1} \bot}\ \text{E-AltF} \qquad \frac{T \Downarrow^{k}_{n} \bot \quad V \in \mathcal{V}}{\sim T \downarrow^{k+1}_{n+1} V}\ \text{E-AltS}$$

$$\frac{}{T \downarrow^{0}_{0} \bot}\ \text{E-Chop}$$

Figure 2: Approximate Evaluation.

$$\frac{\overline{\mathit{fail} \Downarrow \bot} \quad V \in \mathcal{V}}{\epsilon \downarrow V} \qquad \frac{\overline{\mathit{fail} \Downarrow \bot}}{\epsilon \Downarrow \mathcal{V}}$$

$$\frac{\overline{\lambda x.T \downarrow \lambda x.T} \quad \dfrac{V \in \mathcal{V}}{\epsilon \downarrow V} \quad [V/x]T \downarrow \star}{\exists x.\ T \downarrow \star} \qquad \frac{\overline{\lambda x.T \Downarrow \{\lambda x.T\}} \quad \epsilon \Downarrow \mathcal{V} \quad \dfrac{\forall V \in \mathcal{V}.\ ([V/x]T \downarrow \star)}{\forall V \in \mathcal{V}.\ (\sim [V/x]T \Downarrow \bot)}}{\dfrac{\exists x.\ \sim T \Downarrow \bot}{\forall x.\ T \downarrow \star}}$$

Figure 3: Derivations for Quantifier Rules.

Position Paper: Thoughts on Programming with Proof Assistants

Adam Chlipala
Computer Science Division
University of California, Berkeley
Berkeley, California, USA
adamc@cs.berkeley.edu

1 Introduction

Today the reigning opinion about computer proof assistants based on constructive logic (even from some of the developers of these tools!) is that, while they are very helpful for doing math, they are an absurdly heavy-weight solution to use for practical programming. Yet the Curry-Howard isomorphism foundation of proof assistants like Coq [BC04] gives them clear interpretations as programming environments.

My purpose in this position paper is to make the general claim that Coq is already quite useful today for non-trivial certified programming tasks, as well as to highlight some reasons why you might want to consider using it as a base for your next project in dependently-typed programming.

In the last year, I've tried an experiment [Chl06] of using Coq to develop dependently-typed programs of non-trivial size. My application domain has been proof-carrying code, and the idea of "certified program verifiers" in particular. Certified verifiers are essentially an optimization that replaces monolithic proofs of program correctness with re-usable program verifiers that have themselves been proved sound. Almost entirely with Coq, I've implemented a memory safety verifier for x86 machine code programs that use ML-style algebraic datatypes, with a proof of soundness defined in terms of the real x86 machine code semantics.

The remainder of this paper is a summary of what I learned from the experience, starting with a discussion of a short example of the style of code that I found to be effective and concluding with summaries of traditional programming language features that I ended up not missing and Coq features that turned out to be very helpful.

2 Programming with Optional Proofs

Dependent types can be used with inductive type families in very intricate ways. A ubiquitous example is the use of type families for terms of programming languages, where the indices of the type family for a particular meta-language term determine its object-language type. In my work, I've made do with a much more modest subset of the power available in Coq's type system.

I've stuck to dependent types with the flavor of refinement types, where additional logical predicates over values can be attached to standard types. This sort of type naturally describes the result of, for instance, a complete type inference procedure; for any input term, it returns a type known to describe that term. Of course, in most interesting program verification, the properties that we want to check are undecidable, so the type of a procedure like I just described must allow it a way to fail.

The following example illustrates how such types can be used. It uses an even simpler type family, that of optional proofs of a proposition. Let's say that we are writing a verifier that at some point must make sure that a natural number is even before proceeding. Here's Coq code for a function `isEven` to do this:

```
Definition isEven : forall (n : nat), [[even n]].
  refine (fix isEven (n : nat) : [[even n]] :=
```

```
    match n return [[even n]] with
      | 0 => Yes
      | 1 => No
      | S (S n) => pf <- isEven n;
                   Yes
    end); auto.
Defined.
```

The use of this particular form of the `Definition` command indicates that we are constructing this program partially through *interactive proof search*. After the first line, the type of the desired function is asserted as a proof goal, and the body of the definition serves as a proof script to show how to "prove" it, Curry-Howard style. Why not just write the code directly? For this toy example, that works out fine, but, for larger examples, the amount of explicit programming with proofs that this entails is intractable. We would like to take advantage of Coq's features as a proof construction and automation tool as far as possible.

Correspondingly, we begin the body of the definition using the `refine` tactic, which says "I have in mind the structure of the proof, but it has some holes left to fill in." In this case, the structure that we know is the *computational* part of the function, or precisely the part we would write in a setting without formal proofs. The form of the code that we provide gives rise to some remaining proof obligations, and these are queued as subgoals to be handled interactively in usual Coq style.

Now we can turn to the particulars of the definition. The notation `[[`P`]]` denotes the type of optional proofs of proposition P. The `refine` body makes a primitive recursive definition with a `fix` expression, pattern matching on the natural number argument `n`. The notations `Yes` and `No` have the obvious meanings, with only a `Yes` answer registering a new proof obligation. The third, recursive case of the function definition checks the evenness of another natural number before returning its answer. Optional proofs are treated as a *failure monad* in standard Haskell style, enabling the concise notation that I've used. If the recursive call fails, then the current call fails with `No`; while, if the recursive call succeeds, the current call succeeds with `Yes`, and the proof `pf` that is returned can be used in discharging the proof obligation.

The `auto` tactic is chained onto the `refine` tactic with a semicolon, directing Coq to attempt to solve automatically every proof obligation added for the function body. For this simple code, all of the obligations are solvable in this way. In general, the full power of Coq for both user-coded automation and for elaborately scripted manual proof strategies can be used.

The example I've showed uses real Coq syntax, including some user-defined extensions. These extensions are the `[[`P`]]`, `Yes`, `No`, and monadic arrow notations. Expanding these notations and taking into account the results of automated proof search, we get internally an "explicit" definition like the following. One change is that `Yes` has become the constructor `PSome` applied to an explicit proof term. The `auto` tactic probably generates proof terms that aren't as nice to read as those I've used here.

```
Fixpoint isEven (n : nat) : Prop_option (even n) :=
  match n return Prop_option (even n) with
    | O => PSome even_O
    | S O => PNone
    | S (S n) => match isEven n with
                   | PNone => PNone
                   | PSome pf => PSome (even_SS pf)
                 end
  end.
```

3 Pros and Cons of Programming with Coq

3.1 Missing Programming Language Features

Coq is lacking a number of standard programming language features, and several recent language projects [CX05, WSW05] focus on bringing these features to dependently-typed programming.

Two big ones are imperativity and exceptions. I can only say anecdotally that I haven't missed either of these in the work I've described, and I'll point unconvinced readers to their local Haskell enthusiasts for further arguments. I found the failure monad style that I just sketched to be very effective in taking over for one common use of exceptions.

Then there is general recursion or the ability to write non-terminating programs in general. Coq has no separation of logic and programming language, so termination of all terms is required for soundness. I can again mention anecdotally that this only showed up once as a small inconvenience in my work, and Coq has good support for enabling a wide variety of termination arguments.

In summary, there may be areas like "systems" programming where Coq's pure, total programming model is a bad fit, but I believe that it works smoothly in a wide variety of application areas where you might want to bring dependent types to bear. The extent of Coq's programming support would probably surprise most people who haven't use it, as Coq includes a module system, a compilation toolchain that leads to fast native code, and easy integration with normal OCaml code.

3.2 How Coq Supports Effective Programming with Dependent Types

Coq has a number of features designed primarily for "proving" rather than "programming" that nonetheless turn out to be quite useful in dependently typed programming.

- The core of Coq, the Calculus of Inductive Constructions, is very small. This is desirable both because it provides a mathematically elegant and very general solution, and because it brings the trustworthiness benefits of a small trusted code base for a checker for Coq developments. The second point is important in the context of proof-carrying code.
- Languages that separate programming constructs from proof constructs lose the advantages of an idiom called "proof by reflection" [Bou97]. The basic idea of proof by reflection is that checking a proof involves running a program. For instance, in Coq one legal kind of proof of a program's correctness essentially says "Run this certified program verifier and make sure it accepts the program." The proof checker runs the verifier using the same syntactic mechanisms it uses to check proofs in general. It's possible to regain some of these advantages in a language with separate "programming" and "proving" levels by introducing a separate, more pure programming language for use in proofs, but it's nice to avoid this complication.
- Coq's tactic facility makes it easy to script custom decision procedures and use them to construct proofs arising as obligations in dependently-typed programming.
- Coq has a nice ML-style module system for structuring proof developments, programs, and combinations of the two.
- Coq has been around for a while, so there are a lot of libraries, pre-written proof-generating decision procedures, etc., available for it.

4 Conclusion

In the not-so-distant past, Coq was clunky to use and infeasible for real programming. Today, it is mature and reasonable to use for carrying out non-trivial certified programming projects. A number of key features designed originally for formalizing math turn out to play roles in enabling effective dependently-typed programming.

Languages like Epigram [MM04] have very similar foundations to Coq but focus more on programming than proving. The question of which to use seems to hinge on whether "programming" or "proving" aspects dominate the complexity of a program. Many applications in, for instance, high-level programming language semantics involve proofs that closely follow syntax, so that Epigram's features for dependent pattern matching make it a good choice. On the other hand, I think that for cases like reasoning about compilation from high-level languages to machine code, some very large, non-syntax-directed proofs will inevitably be involved, so that the biggest productivity gains are to be had by taking advantage of some serious proof organization and automation machinery. It also seems a promising direction to investigate the best ways of importing some of Coq's more recent proof-oriented features into more traditional programming settings.

References

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[Bou97] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *Theoretical Aspects of Computer Software*, pages 515–529, 1997.

[Chl06] Adam Chlipala. Modular development of certified program verifiers with a proof assistant. In *ICFP '06: Conference record of the 11th ACM SIGPLAN International Conference on Functional Programming*, September 2006. To appear.

[CX05] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 66–77, 2005.

[MM04] Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004.

[WSW05] Edwin Westbrook, Aaron Stump, and Ian Wehrman. A language-based approach to functionally correct imperative programming. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 268–279, 2005.