Coding Binding and Substitution Explicitly in Isabelle

Christopher Owens

Laboratory for Foundations of Computer Science,
Department of Computer Science, King's Buildings,
The University of Edinburgh, EH9 3JZ, UK.

Abstract

Logical frameworks provide powerful methods of encoding object-logical binding and substitution using meta-logical λ-abstraction and application. However, there are some cases in which these methods are not general enough: in such cases object-logical binding and substitution must be explicitly coded. McKinna and Pollack [MP93] give a novel formalization of binding, where they use it principally to prove meta-theorems of Type Theory. We analyse the practical use of McKinna-Pollack binding in Isabelle object-logics, and illustrate its use with a simple example logic.

1 Introduction

In this paper we address the problem of coding logics in Isabelle (and other logical frameworks). In some logics, meta-logical binding is unsuitable for coding object-logical binding. For example, as we explain later, logics for programming languages which include features such as pattern matching are difficult to code using meta-binding. We advocate the use of the binding system due to McKinna and Pollack [MP93] to code binding explicitly in these cases. The aim of this paper is to give enough meta-theoretical background and practical examples to facilitate the coding of a logic in Isabelle using McKinna-Pollack binding, and to motivate and evaluate the choice of McKinna-Pollack binding.

In the remainder of this section, we discuss some motivating examples, and compare systems which allow us to code binding and substitution explicitly, explaining why we choose the system due to McKinna and Pollack. In Section 2 we describe McKinna-Pollack binding and give an example of its use, together with several meta-theorems which capture its behaviour. In Section 3 we expand on the example, and discuss features of the implementation of the system in Isabelle. In Section 4, we return to our motivating concern and examine how McKinna-Pollack binding can encode logics for programming languages. Finally, in Section 5, we summarize our results and evaluate the possible uses of McKinna-Pollack binding style in Isabelle.

Most Isabelle code is omitted from this document. However, this should not disguise the fact that this paper is about an Isabelle implementation. The logic is built on Isabelle HOL. When we give a grammar, it is implemented in Isabelle as the obvious datatype

declaration. Similarly, inductive definitions are coded in the obvious way in Isabelle, as are primitive recursive definitions. Where the translation from the page to Isabelle is not immediate, the implementation is explained. The Isabelle code is available by ftp from `ftp.dcs.ed.ac.uk` in directory `/home/ftp/pub/cao/IUW`, and on the world-wide web as `http://www.dcs.ed.ac.uk/home/cao/IUW`.

1.1 Motivation

Much research has been devoted to coding logics in logical frameworks and generic theorem provers [PE88, DFH95]. Most of these encodings use meta-level λ-abstraction to encode object-level binding, meta-application to encode object-substitution, and meta-variables to represent object variables. However, some logics can be difficult or cumbersome to encode in this manner. In particular, difficulties with logics for programming languages arise because of their sophisticated binding and scoping features.

Throughout the rest of this paper we will use an encoding of a version of the Simple Theory of Types [Chu40] to provide examples. Judgements are of the form $\Gamma \vdash P$, where Γ is a list of declarations either of the form $v\colon \tau$ (variable v has type τ) or $v\colon \sigma = M$ (variable v is term M of (polymorphic) type σ). Declarations may make reference to earlier declarations, for example in the context $x\colon \tau \to \tau\,;\; y\colon \tau \to \tau = \lambda b\colon \tau.\, xb$, the declaration of y refers to the earlier declaration of x.

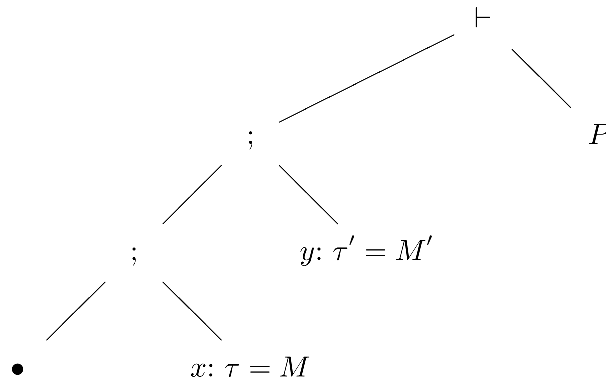
Consider this judgement:

$$x\colon \tau = M\,;\; y\colon \tau' = M' \vdash P$$

If we were to use meta-binding to represent the binding of x and y, we might encode this judgement in the following way:

$$\mathsf{ValDec}(M, \lambda x.\, \mathsf{ValDec}(M', \lambda y.\, \mathsf{Judge}(P)))$$

However, as we shall see, type-checking rules and rules of the logic are only concerned with the last declaration in the context, which in this coding is the inner-most declaration: it is not clear how to express such rules with this coding. We would like to represent the judgement as the obvious abstract syntax tree (the empty context is written $\bullet$):



In order to use this scheme, we must encode the binding that occurs in declarations explicitly. Adding further scoping features—let-expressions, local declarations, let rec declarations—accentuates the problem.

Now we show that we cannot use meta-binding to bind variables *within* terms. Consider a logic for a language which includes pattern matching, having a construct such as the Standard ML `case` expression. Each pattern can bind an arbitrary number of variables, and so we must "compile" a case-expression into some internal form, where each variable is bound separately. Let us take as an example the following expression:

$$\texttt{case}\ e\ \texttt{of (}x\texttt{, }y\texttt{) => }x\texttt{ + }y$$

This might compile into the following form:

$$\mathsf{CaseExp}(e, \mathsf{PatBind}(\lambda x.\, \mathsf{PatBind}(\lambda y.\, \mathsf{PatAndExp}(\texttt{(}x\texttt{, }y\texttt{), }x\texttt{ + }y))))$$

This translation would be done by print- and parse-translations. The same ML expression is also represented by another compiled form:

$$\mathsf{CaseExp}(e, \mathsf{PatBind}(\lambda y.\, \mathsf{PatBind}(\lambda x.\, \mathsf{PatAndExp}(\texttt{(}x\texttt{, }y\texttt{), }x\texttt{ + }y))))$$

There are two objections to representing pattern matching in this manner:

- Although the two compiled forms above would appear the same after print translation, they cannot be unified. Furthermore, we must add rules which equate the different compiled forms of an expression.
- The parse- and print-translations are difficult to program. More importantly, it is very difficult to establish that they have been programmed correctly.

These problems arise from the fact that pattern matching should be a *single* binding which binds many variables at once. This is not the same as multiple bindings of single variables. With such a "compiled" coding, it is difficult to satisfy ourselves that the logic actually represents expressions as we intend.

By coding binding explicitly, we can represent `case`-expressions just as the grammar of the language suggests:

$$\mathsf{CaseExp}(e, \mathsf{Match}(\texttt{(}x\texttt{, }y\texttt{), }x\texttt{ + }y))$$

The translations are trivial, each syntactically distinct expression has a unique representation, and the correctness of the representation is a non-issue.

1.2 Systems of Explicit Binding and Substitution

Rather than using increasingly arcane and counter-intuitive codings, our approach is to axiomatize binding. We consider three possible systems.

De Bruijn binding [dB72] eliminates α-conversion by using indices rather than names for bound variables—terms which are α-variants in name-carrying systems are syntactically identical in de Bruijn systems. De Bruijn binding is widely used, for example in Type Theory (for example by Altenkirch [Alt93]), and in theorem provers (including Isabelle [Pau94a]).

Unfortunately, de Bruijn terms are very difficult to read, and they are usually translated to and from a name-carrying form for display and input. This can be done by providing a software front-end to perform the translation, but Gordon [Gor93] encodes

the translation within the framework of the theorem prover. However, any method of translation adds another layer of processing on top of the naked de Bruijn terms: this adds complexity and reduces clarity and abstraction. It is also the case that many formal systems are defined using names (for example, Standard ML [MTH90]), and we should be wary of the faithfulness of an encoding of a logic based on such a formal system if the encoding does not use names.

The "standard" method of binding is originally due to Curry and Feys [CF58] and is given a clear exposition by Hindley and Seldin [HS86]. Standard binding systems define the substitution $(\lambda x.\, M)[N/y]$ as $\lambda z.\, M[z/x][N/y]$ for some fresh z (at least in the case that capture would occur). Stoughton [Sto88] observes that since $M[z/x]$ is not a sub-term of $\lambda x.\, M$, substitution cannot be defined by recursion on the structure of terms, and must instead be defined by recursion on the size of terms. This complicates proof considerably. He proposes a system which performs the substitutions in parallel, $M[z/x, N/y]$, which means that substitution *can* then be defined by recursion on the structure of terms, since M certainly is a sub-term of $\lambda x.\, M$.

We regard this difficulty in defining substitution as a clue that the substitution operation is the wrong place to worry about variable capture, and that capture should instead be avoided in the construction of the terms, allowing substitution to be defined simply. The system used in this paper is due to McKinna and Pollack [MP93]: it uses names for bound variables. The theory of this style of binding is further developed by McKinna [McK]. A side-effect of this care in term construction is that α-conversion is rarely necessary: it comes "for free".

2 McKinna-Pollack Binding

McKinna and Pollack give a system of binding and substitution in which variables are divided into two disjoint classes, intended to represent free and bound variables. The two classes of variable are referred to as *f-variables* (f, f', intended to represent free variables—McKinna and Pollack call these parameters) and *b-variables* (b, b', representing bound variables—called simply variables by McKinna and Pollack). It is syntactically impossible for an f-variable to be bound in a term, and it is an important meta-theorem (to be formalized later as Theorem 7) that no b-variable occurs free in a valid deduction.

In McKinna-Pollack binding, both classes of variables are names and not de Bruijn indices. Specifically, all we know about them is that there are infinitely many of them (in the sense that one can always pick a fresh one). Typically, f- and b-variables will be picked from the same set of identifiers, with a constructor wrapped around them. Isabelle itself splits variables into free and bound variables [Pau94b, Section 6.5], but in Isabelle, bound variables are represented by de Bruijn indices.

The grammar for simply-typed lambda terms which we will use in our example logic is given in Figure 1. We assume that the set of constants is disjoint from the sets of b- and f-variables. We write object equality as "$\approx$" in order to differentiate it from Isabelle meta-equality ($\equiv$) and Isabelle HOL equality ($=$). Hilbert's description operator ("any") is written ε. The type Ω is the type of truth-values. Terms with no free b-variables are called *b-closed*.

$$
\begin{array}{lll}
\mathit{ty} & ::= & \Omega \\
 & & \mathit{ty}_1 \rightarrow \mathit{ty}_2 \\
 & & \\
\mathit{constant} & ::= & \supset \\
 & & \varepsilon \\
 & & \approx \\
 & & \\
\mathit{term} & ::= & \mathit{fvar} \\
 & & \mathit{bvar} \\
 & & \mathit{constant} \\
 & & \lambda \mathit{bvar}\colon \mathit{ty}.\, \mathit{term} \\
 & & \mathit{term}_1\ \mathit{term}_2
\end{array}
$$

Figure 1: simply-typed lambda terms.

$$
\begin{array}{llcl}
(\mathit{BSubstF}) & f[N/b] & = & f \\
(\mathit{BSubstB}) & b'[N/b] & = & \mathrm{if}\,(b = b',\ N,\ b') \\
(\mathit{BSubstC}) & \mathit{const}[N/b] & = & \mathit{const} \\
(\mathit{BSubstLda}) & (\lambda b'\colon \tau.\, M)[N/b] & = & \mathrm{if}\,(b = b',\ \lambda b'\colon \tau.\, M,\ \lambda b'\colon \tau.\, M[N/b]) \\
(\mathit{BSubstApp}) & (M\, M')[N/b] & = & M[N/b]\, M'[N/b]
\end{array}
$$

Figure 2: b-substitution.

2.1 Substitution and Closedness

There are two notions of substitution. Substitution of a term for a b-variable (b-substitution, written $M[N/b]$) is defined by primitive recursion on terms by the rules in Figure 2. Since f-variables, b-variables and constants are disjoint, b-substitution at f-variables and constants does nothing. As in standard substitution, a binding of a variable will shadow substitution for it (the rule *BSubstLda* with $b = b'$). However, unlike standard substitution, we do not α-convert to avoid capture in the substituted term (*BSubstLda* with $b \neq b'$). This is because we know that a b-closed term has no free b-variables—and, in particular, the variable bound by the λ-abstraction is not free in the substituted term.

Substitution of a term for an f-variable (f-substitution, written $M[f = N]$) is defined by primitive recursion on terms by the rules in Figure 3. This is simply textual substitution: everywhere the f-variable f appears in M, it is replaced by N. In particular, substitution at λ-abstractions does not need to take into account shadowing of the substitution by the binding: since λ can only bind b-variables, it can never shadow a substitution for an f-variable. We can think of f-variables as "holes" in terms, and f-substitution $M[f = N]$ as plugging a term N into all the holes labelled f in M.

The following theorem shows the connection between the two kinds of substitution. We write $b \in M$ to mean that b-variable b occurs free in M, and $f \in M$ to mean that f-variable f occurs in M—effectively meaning "occurs *free*" because f-variables can

$$
\begin{array}{llcl}
(FSubstF) & f'[f=N] & = & \text{if}\,(f=f',\,N,\,f') \\
(FSubstB) & b[f=N] & = & b \\
(FSubstC) & const[f=N] & = & const \\
(FSubstLda) & (\lambda b\colon \tau.\, M)[f=N] & = & \lambda b\colon \tau.\, M[f=N] \\
(FSubstApp) & (M\, M')[f=N] & = & M[f=N]\; M'[f=N]
\end{array}
$$

Figure 3: f-substitution.

$$
(BClosedF) \qquad \frac{}{f \in \mathsf{BClosed}}
$$

$$
(BClosedC) \qquad \frac{}{const \in \mathsf{BClosed}}
$$

$$
(BClosedLda) \qquad \frac{M[f/b] \in \mathsf{BClosed} \quad f \notin M}{\lambda b\colon \tau.\, M \in \mathsf{BClosed}}
$$

$$
(BClosedApp) \qquad \frac{M \in \mathsf{BClosed} \quad M' \in \mathsf{BClosed}}{M\, M' \in \mathsf{BClosed}}
$$

Figure 4: the set BClosed.

never be bound in terms. The formal definitions are left to the interested reader: they are very simple.

Theorem 1 (Factorisation.) *Given $f \notin M$:*

$$M[N/b] = M[f/b][f = N]$$

Proof: By induction on M.

□

We are now in a position to give a formal definition of b-closed terms. The set BClosed is given inductively by the rules in Figure 4.

In the rule *BClosedLda* we require $f \notin M$: this is the usual form for rules in the McKinna-Pollack style. We are careful to maintain b-closedness of terms (free occurrences of b have been substituted away in $M[f/b]$). The side-condition ensures that b is not identified with any f-variable occurring in M by the substitution $M[f/b]$—this would be a form of variable capture.

In this case we are only concerned with the b-closedness of the term, and so do not care if f and b are identified. We can define BClosed′ in exactly the same way as BClosed, but omitting the side-condition on the λ rule (McKinna and Pollack use this definition of b-closedness). The sets BClosed and BClosed′ can then be proved identical by a messy induction on the size of terms. In general, however, when we define a relation we wish to avoid this kind of capture, and so the side-condition cannot be omitted.

Theorem 2 (Substitution preserves b-closedness.)

1. *For any f-variable f, if M and N are b-closed terms, then $M[f = N]$ is b-closed.*

2. *If $\lambda b\colon \tau.\, M$ and N are b-closed terms, then $M[N/b]$ is b-closed.*

The converse of neither part of this theorem holds: if $f \notin M$, then $M[f = N] = M$, which tells us nothing about N, and similarly if $b \notin M$, $M[N/b] = M$.

An attempt to prove this theorem by induction on the definition of $\mathsf{BClosed}$ will fail. This is because in the rule *BClosedLda*, we only require that there *exists* a suitable f. The induction hypothesis for the λ case tells us that there is an f' such that $M[f'/b][f = N] \in \mathsf{BClosed}$, but we are attempting to prove $M[f = N][f'/b] \in \mathsf{BClosed}$ for *every* f'. This problem is common when proving properties of McKinna-Pollack-style relations.

Intuitively, since $M[f/b]$ is b-closed for some fresh f, it is in fact b-closed for every possible f. We define the set $\mathsf{BClosed}''$ in the same way as $\mathsf{BClosed}$, but with the following λ rule:

$$(BClosedBLda)\quad \frac{\forall f. M[f/b] \in \mathsf{BClosed}''}{\lambda b\colon \tau.\, M \in \mathsf{BClosed}''}$$

Surprisingly, it is not necessary to require f to be fresh. The quantifier in the hypothesis of this rule means that $\mathsf{BClosed}''$ cannot (currently) be declared as an Isabelle inductive set: it must be hand-coded as a least fixed point. We now show that the sets $\mathsf{BClosed}''$ and $\mathsf{BClosed}$ are identical. It is immediate that $\mathsf{BClosed}'' \subseteq \mathsf{BClosed}$, since if the hypothesis holds for every f, there certainly exists a fresh f such that it holds. To prove $\mathsf{BClosed} \subseteq \mathsf{BClosed}''$, we must introduce the notion of *renaming*. A renaming is a finite map from f-variables to f-variables. We lift renamings ρ to be operations on terms, $\rho^*(_)$, in the obvious way.

Theorem 3 *If $M \in \mathsf{BClosed}$, then, for all renamings ρ, $\rho^*(M) \in \mathsf{BClosed}''$.*

Proof: By induction on the definition of $\mathsf{BClosed}$. All the cases are simple, except the λ case. This case boils down to deducing $\rho_0^*(M)[f_0/b] \in \mathsf{BClosed}''$ for arbitrary ρ_0, f_0. The induction hypothesis is $\forall \rho.\, \rho^*(M[f'/b]) \in \mathsf{BClosed}''$, where $f' \notin M$. We instantiate ρ in the induction hypothesis with $\rho_0 + \{f' \mapsto f_0\}$, and the result follows by equational reasoning.

□

Corollary 4

$$\mathsf{BClosed} = \mathsf{BClosed}''$$

Proof: We already know $\mathsf{BClosed}'' \subseteq \mathsf{BClosed}$. We can deduce $\mathsf{BClosed} \subseteq \mathsf{BClosed}''$ from the previous theorem with $\rho = \emptyset$.

□

Such proofs are generally possible for relations defined in McKinna-Pollack style. We can now use the induction rule for $\mathsf{BClosed}''$ to prove properties of $\mathsf{BClosed}$, as required.

Finally, we are in a position to prove Theorem 2.

Proof of Theorem 2: Part 1 is an induction on $\mathsf{BClosed}''$. It requires the following lemma:

$$N' \in \mathsf{BClosed} \supset M[N/b][f = N'] = M[f = N'][N[f = N']/b]$$

In turn, this requires the lemma $N \in \mathsf{BClosed} \supset M[N/b] = M$.

Part 2 follows from part 1 by the Factorisation Theorem (Theorem 1).

□

We show that $\mathsf{BClosed}$ correctly formalizes the set of b-closed terms.

Theorem 5 *Given a term M, $M \in \mathsf{BClosed}$ if and only if $b \notin M$ for every b-variable b.*

Proof: This is proved by induction on the size of M.

□

We have no further use for the test $b \in M$, and only use $\mathsf{BClosed}$ from now on. Before moving on, note that we could have defined $b \in M$ to be $f \in M[f/b]$, where $f \notin M$.

2.2 Type-checking and Contexts

Having developed the machinery of McKinna-Pollack substitution, we move on to a first application: type-checking.

The type system is an adaptation of HOL's. The grammar for types allows only monomorphic types: generalised types such as $A \to A$ (the type of functions from any type to itself) are admitted by using Isabelle free meta-variables to stand for types, (so A is an Isabelle free meta-variable in $A \to A$). The constants are genuinely polymorphic; that is, they may have more than one type.

Declarations of terms are also polymorphic. They are written $f\colon \forall\alpha_1, \ldots, \alpha_n.\, \tau = M$. This indicates that M may have any type of the form τ' where τ' is τ with each α_i replaced by a type. The variables α_i may also appear in M, as in the declaration $f\colon \forall\alpha.\, \alpha \to \alpha = \lambda b\colon \alpha.\, b$. In fact, then, $f\colon \forall\alpha_1, \ldots, \alpha_n.\, \tau = M$ is simply syntactic sugar for $\mathsf{Val}(f, S)$ where S is the *declaration scheme* $\forall\alpha_1, \ldots, \alpha_n.\, \langle\langle\tau, M\rangle\rangle$. Binding of the α_i in declaration schemes is coded as Isabelle meta-binding in the usual Isabelle way. Declaration schemes are formed from two constructors: basic schemes are just types paired with a term of that type; abstracted schemes are schemes quantified over a variable:

```
BasicSch    :: "[Ty, Term] => Scheme"       ("<<_, _>>")
AbsSch      :: "(Ty => Scheme) => Scheme"   (binder "SCH " 100)
```

The relation $\mathsf{InstType}(\forall\alpha_1, \ldots, \alpha_n.\, \langle\langle\tau, M\rangle\rangle, \tau')$ holds when τ' can be obtained from τ by instantiating the variables $\alpha_1, \ldots, \alpha_n$. Similarly, $\mathsf{InstTerm}(\forall\alpha_1, \ldots, \alpha_n.\, \langle\langle\tau, M\rangle\rangle, M')$ holds when M' can be obtained from M by instantiating $\alpha_1, \ldots, \alpha_n$. Their definitions are omitted, but are simple.

Declarations of types for f-variables are written $f\colon \tau$: the type τ is not polymorphic. Contexts are lists of declarations of either form. Notice that contexts declare terms and types for f-variables, not b-variables.

The typing judgement is defined inductively by the rules in Figure 5.

$$\begin{array}{lcl} \mathsf{TypesOf}(\supset) & = & \{\Omega \to \Omega \to \Omega\} \\ \mathsf{TypesOf}(\varepsilon) & = & \{(A \to \Omega) \to A, A \text{ is a type}\} \\ \mathsf{TypesOf}(\approx) & = & \{A \to A \to \Omega, A \text{ is a type}\} \end{array}$$

$$(\mathit{LookupVbl}) \qquad \frac{}{\tau \in \mathsf{LookupTypes}(\Gamma;\ f\colon \tau, f)}$$

$$(\mathit{LookupVal}) \qquad \frac{\mathsf{InstType}(S, \tau)}{\tau \in \mathsf{LookupTypes}(\Gamma;\ \mathsf{Val}(f, S), f)}$$

$$(\mathit{LookupWeak}) \qquad \frac{\tau \in \mathsf{LookupTypes}(\Gamma, f)}{\tau \in \mathsf{LookupTypes}(\Gamma;\ \mathit{dec}, f)} \qquad f \text{ not declared by } \mathit{dec}$$

$$(\mathit{TypF}) \qquad \frac{\tau \in \mathsf{LookupTypes}(\Gamma, f)}{\Gamma \vdash f\colon \tau}$$

$$(\mathit{TypC}) \qquad \frac{\tau \in \mathsf{TypesOf}(\mathit{const})}{\Gamma \vdash \mathit{const}\colon \tau}$$

$$(\mathit{TypLda}) \qquad \frac{\Gamma;\ f\colon \tau \vdash M[f/b]\colon \tau' \quad f \notin M}{\Gamma \vdash \lambda b\colon \tau.\, M\colon \tau \to \tau'}$$

$$(\mathit{TypApp}) \qquad \frac{\Gamma \vdash M\colon \tau \to \tau' \quad \Gamma \vdash M'\colon \tau}{\Gamma \vdash M\, M'\colon \tau'}$$

Figure 5: the typing judgement.

Now we define what it means for a context Γ to be valid, written $\vdash \Gamma$. We say a declaration $f\colon \forall\alpha_1 \ldots \alpha_n.\, \tau = M$ is well-typed in Γ if and only if for every instance τ' of τ and corresponding instance M' of M, $\Gamma \vdash M'\colon \tau'$. Declarations of the form $f\colon \tau$ are always considered to be well-typed. A context $\Gamma;\ \mathit{dec}$ is valid exactly when Γ is valid and dec is well-typed in Γ (the empty context is valid, too, of course).

The typing rules only deduce a type for b-closed terms. This is formalized as the following theorem.

Theorem 6

1. *If* $\Gamma \vdash M\colon \tau$ *then* $M \in \mathsf{BClosed}$.
2. *If* $\vdash \Gamma$ *then, for every declaration of the form* $f\colon \forall\alpha_1 \ldots \alpha_n.\, \tau = M$ *occurring in* Γ, $M \in \mathsf{BClosed}$.

Proof:

1. By induction on the definition of the typing judgement.
2. A simple corollary of part 1, since $\vdash \Gamma$ means that all terms in Γ are well-typed.

□

3 A Version of the Simple Theory of Types

The version of the Simple Theory of Types we implement is similar to that implemented in HOL [GM93] and to an early version of Isabelle HOL [Pau90]. It contains both λ-binding and also binding of terms in declarations within a context, and so illustrates the use of McKinna-Pollack binding.

Judgements in the logic are of the form $\Gamma \vdash P$, meaning "P holds in the presence of context Γ." Recall that contexts give types to f-variables and declarations of values for f-variables. We ensure type-correctness of judgements by ensuring that axioms are well typed, and that rules preserve well-typedness, as is usual.

The rules and axioms of the system are given in Figure 6. Rules whose only hypotheses are well-formedness conditions are considered to be axioms.

3.1 Declarations

We can look-up declarations via the rule *Lookup*:

$$\frac{\Gamma \vdash M \approx M' \quad f \notin M' \quad \mathsf{InstTerm}(S, M)}{\Gamma;\ \mathsf{Val}(f, S) \vdash f \approx M'}$$

Notice that M is evaluated in the context Γ: any occurrences of f in M refer to a previous declaration in Γ.

We give an atomic weakening rule for contexts. The declaration *dec* is a single value or type declaration which declares the f-variable f.

$$\frac{\Gamma \vdash P \quad \mathit{dec} \text{ well-typed in } \Gamma \quad f \notin P}{\Gamma;\ \mathit{dec} \vdash P}$$

This rule is provably equivalent to weakening by a many declarations at once.

The logical connectives T, F, $\wedge$, $\vee$, $\neg$, $\forall$ and $\exists$ are declared in a standard context which gives their usual HOL definitions. The example proofs given in the `.ML` files culminate in the derivation of the usual natural deduction rules for these connectives.

3.2 Equality and Conversion

We encode β-reduction in the usual HOL way: we say that two terms are equal if one is obtained from the other by reducing a single outer-most β redex. The other rules for equality mean that it includes the reflexive transitive closure of β-reduction within terms.

McKinna-Pollack binding style does not require us to explicitly define α-conversion. As a side-effect, the rule *Abs* captures α-conversion. For example, $\Gamma \vdash \lambda x{:}\ \tau.\, x \approx \lambda y{:}\ \tau.\, y$, by *EqRefl* followed by *Abs*. (Of course, α-conversion is admitted anyway in the Simple Theory of Types by extensionality, but even in a system where this were not so, α-conversion would still be admitted by *Abs*).

The following is a simple consequence of the axiom *EtaConv*:

$$\frac{\vdash \Gamma \quad \Gamma \vdash H{:}\ \tau \to \tau'}{\Gamma \vdash (\lambda b{:}\ \tau.\, H\, b) \approx H}$$

In a standard binding system this rule would have the additional side-condition $b \notin H$—but since H is well-typed, it is b-closed, and so we need not check this.

Axioms

(*EqRefl*) $$\frac{\vdash \Gamma \quad \Gamma \vdash M\colon \tau}{\Gamma \vdash M \approx M}$$

(*BetaConv*) $$\frac{\vdash \Gamma \quad \Gamma \vdash (\lambda b\colon \tau.\, M)\, N\colon \tau}{\Gamma \vdash ((\lambda b\colon \tau.\, M)\, N) \approx M[N/b]}$$

(*OmegaCases*) $$\frac{\vdash \Gamma}{\Gamma \vdash \forall P\colon \Omega.\, (P \approx \mathsf{T}) \vee (P \approx \mathsf{F})}$$

(*ImpAntiSym*) $$\frac{\vdash \Gamma}{\Gamma \vdash \forall P, Q\colon \Omega.\, (P \supset Q) \supset (Q \supset P) \supset (P \approx Q)}$$

(*EtaConv*) $$\frac{\vdash \Gamma}{\Gamma \vdash \forall H\colon \tau \to \tau'.\, (\lambda b\colon \tau.\, H\, b) \approx H}$$

(*Select*) $$\frac{\vdash \Gamma}{\Gamma \vdash \forall P\colon \alpha \to \Omega.\, \forall x\colon \alpha.\, P\, x \supset P\, (\varepsilon\, P)}$$

Rules

(*Subst*) $$\frac{\Gamma \vdash P[f = M] \quad \Gamma \vdash M \approx M'}{P[f = M']}$$

(*Abs*) $$\frac{\Gamma;\ f\colon \tau \vdash M[f/b] \approx M[f/b'] \quad f \notin M, M'}{\Gamma \vdash \lambda b\colon \tau.\, M \approx \lambda b'\colon \tau.\, M'}$$

(*ImpI*) $$\frac{\begin{array}{c}[\Gamma \vdash P] \\ \vdots \\ \Gamma \vdash Q\end{array}}{\Gamma \vdash P \supset Q}$$

(*MP*) $$\frac{\Gamma \vdash P \quad \Gamma \vdash P \supset Q}{\Gamma \vdash Q}$$

(*Lookup*) $$\frac{\Gamma \vdash M \approx M' \quad f \notin M' \quad \mathsf{InstTerm}(S, M)}{\Gamma;\ \mathsf{Val}(f, S) \vdash f \approx M'}$$

(*Weak*) $$\frac{\Gamma \vdash P \quad \mathit{dec} \text{ well-typed in } \Gamma \quad f \notin P}{\Gamma;\ \mathit{dec} \vdash P} \qquad \mathit{dec} \text{ declares } f$$

Figure 6: The Simple Theory of Types

3.3 Deductions

Unfortunately, the judgement $\Gamma \vdash P$ cannot be defined inductively, because of its negative occurrence in the discharged hypothesis in *ImpI*. The only practical problem this causes us is the inability to prove meta-theorems by induction on the definition of $\Gamma \vdash P$ within Isabelle. Informally, we can still do induction on *derivations*—and we could formalize this, as Paulson has done for his proof of the completeness of propositional logic in the Isabelle examples library, but the system would not be usable for real proof.

Theorem 7 *(Well-formedness of deductions) Suppose we deduce the following:*

$$\frac{\Gamma_1 \vdash P_1 \quad \dots \quad \Gamma_n \vdash P_n}{\Gamma \vdash P}$$

1. *If $\Gamma_i \vdash P_i \colon \Omega$ for every i, then $\Gamma \vdash P \colon \Omega$.*
2. *If $\vdash \Gamma_i$ for every i, then $\vdash \Gamma$.*
3. *If $P_i \in \mathsf{BClosed}$ for every i, then $P \in \mathsf{BClosed}$.*
4. *If $M \in \mathsf{BClosed}$ for every term M in a Γ_i, then for every M appearing in Γ, $M \in \mathsf{BClosed}$.*

Proof: By inspection, each of these properties holds for each of the rules and axioms in Figure 6. Hence, by induction on derivations, they hold for all derivations.

□

Corollary 8 1. *Suppose we deduce the following:*

$$\frac{\Gamma_1 \vdash P_1 \quad \dots \quad \Gamma_n \vdash P_n}{\Gamma \vdash P}$$

If $\Gamma_i \vdash P_i \colon \Omega$ for every i, then $\Gamma \vdash P \colon \Omega$, $\vdash \Gamma$, $P \in \mathsf{BClosed}$, and $M \in \mathsf{BClosed}$ for every M occurring in Γ.

2. *Property 1 and the properties in Theorem 7 each hold for every node $\Gamma \vdash P$ in a derivation tree.*

Proof:

1. Since $\Gamma_i \vdash P_i \colon \Omega$, then $\vdash \Gamma_i$, $P_i \in \mathsf{BClosed}$ and $M \in \mathsf{BClosed}$ for every M occurring in a Γ_i, and therefore $\Gamma \vdash P \colon \Omega$, $\vdash \Gamma$, $P \in \mathsf{BClosed}$, and $M \in \mathsf{BClosed}$ for every M occurring in Γ.
2. To see that these properties hold of interior nodes of a derivation tree, observe that each interior node is itself the conclusion of a sub-derivation of the tree.

□

3.4 Implementation

The Isabelle implementation raises issues relating to giving proofs *using* the logic, rather than simply meta-proofs *about* the logic.

The implementation of the substitutivity rule illustrates some of these issues. The usual McKinna-Pollack coding substitutivity is given in Figure 6. The term P has some "holes" in it (the f-variable f); these holes are filled by M in the antecedent and M' in the conclusion. Since we are using Isabelle, however, we can use meta-substitution in this case: a term P with some holes filled by M is $P(M)$ in Isabelle. Since the logic does not use meta-binding, this is simply textual substitution, as required. The rule becomes:

$$\frac{\Gamma \vdash M \approx M' \quad \Gamma \vdash P(M)}{\Gamma \vdash P(M')}$$

There is also a problem with the applicability of some rules in their current form. Consider attempting to deduce the following.

$$\Gamma \vdash ((\lambda b\colon \tau.\, b) f) \approx f$$

This is simply a β-reduction. However, we cannot apply the *BetaConv* axiom, since f is not in the form $b[f/b]$. Instead, we must first derive the following rule, which is applicable.

$$\frac{M[N/b] = M' \quad \vdash \Gamma \quad \Gamma \vdash (\lambda b\colon \tau.\, M)\, N\colon \tau}{\Gamma \vdash ((\lambda b\colon \tau.\, M)\, N) \approx M'}$$

When this rule is applied, we obtain a subgoal $b[f/b] = f$, which can be solved by the simplifier.

Now consider performing the substitution $b'[N/b]$. In the case that b and b' are identical, this can be solved immediately to yield N. Suppose b and b' are not identical (and are intended to be distinct). Since b and b' are both implicitly universally quantified meta-variables, $b \neq b'$ cannot in general be deduced, and so we cannot proceed any further and must leave the substitution in the form $\mathrm{if}\,(b = b',\, N,\, b')$. This is a consequence of the use of meta-variables to represent object-variables. We must add $b \neq b'$ as an assumption.

In fact, the neatest way to do this is to define a predicate $\mathsf{distinct}$, which says that the elements of a list are pair-wise distinct. We now simply add the assumption $\mathsf{distinct}(bs)$ to all proofs, where bs is a list containing all those b-variables which appear in the proof. We must similarly provide the assumption $\mathsf{distinct}(fs)$.

In the theory files accompanying this paper, f-variables are formed either from identifiers or from one of the connectives T, F, $\wedge$, $\vee$, $\neg$, $\forall$ and $\exists$. This means that the connectives can be proved to be distinct from each other and all other f-variables, without needing to assert $\mathsf{distinct}(\mathsf{T}, \mathsf{F}, \wedge, \vee, \neg, \forall, \exists, \ldots)$.

Not all of Isabelle's automatic proof procedures may be used with this logic. No work has yet been performed on automating proof, but we believe it is possible to set up `hyp_subst_tac` for this logic. Neither the new simplifier nor the old simplifier can be used with this logic (because of the side-condition on the reflexivity axiom *EqRefl*), however, it should be possible to use the "typed simplifier" `TSimpFun`, which is used in Isabelle's Constructive Type Theory. It may also be possible to configure the classical reasoning package to work with this logic.

4 Programming Languages

In the Introduction we showed that logics for programming languages are difficult to code using meta-binding. We now give examples of how they can be implemented using McKinna-Pollack binding. The features for which we give an account here (let expressions, local declarations, and pattern matching) can be found in many programming languages. We consider a simple eager statically-bound language with ML-like syntax.

The logic we outline is based on the logic of Section 3. Logical terms may now include programming language expressions. We also allow the context to contain programming language declarations. This means that a judgement in the logic would typically be a statement about programming language expressions in the presence of programming language declarations.

4.1 Let-Expressions and Local Declarations

Let-expressions and local declarations allow us to limit the scope of certain declarations. Let-expressions are of the form:

$$\texttt{let}\ \mathit{decs}\ \texttt{in}\ e\ \texttt{end}$$

Here, *decs* is a list of declarations. It differs from the list of declarations in the context in two ways:

- It declares values for b-variables, not f-variables. McKinna-Pollack binding relies on the fact that no f-variable binding occurs within a b-variable binding, and a let-expression might occur within another construct that binds b-variables (a λ-abstraction, for instance).
- We will primarily be concerned with the left-most declaration in *decs*. So for convenience, we insist that the list associates to the right: the left-most declaration is the outer-most.

The rules for let-expressions allow declarations to be moved to and from the context. They are complicated by the fact that local declarations will mean that a single declaration can bind more than one variable.

One such complication is that we will need a multiple b-substitution operation, $X[f_1/b_1, \ldots, f_n/b_n]$. We only use this operation to substitute f-variables for b-variables, and so we can perform the substitutions of f_i for b_i in any order (or in parallel) since they cannot affect each other.

The operation $\mathit{dec}\langle f_1/b_1, \ldots, f_n/b_n\rangle$ replaces binding occurrences of b_i by f_i, and replaces bound occurrences of b_i with f_i. For example:

$$\begin{array}{c}(\texttt{val}\ b_j\ \texttt{=}\ e\texttt{;}\ \mathit{decs})\langle f_1/b_1, \ldots, f_n/b_n\rangle \\ = \\ \texttt{val}\ f_j\ \texttt{=}\ e\texttt{;}\ \mathit{decs}[f_j/b_j]\langle f_1/b_1, \ldots, f_n/b_n\rangle\end{array}$$

In order to avoid variable capture, we require that $f_i \neq f_j$ when $b_i \neq b_j$. Thus $\mathit{dec}\langle f_i/b_i\rangle$ is a declaration which declares f-variables f_1, ..., f_n and which has exactly the same shape as *dec*.

Let us write $X\langle f_1/b_1, \ldots, f_n/b_n\rangle$ as $X\langle f_i/b_i\rangle$ and $X[f_1/b_1, \ldots, f_n/b_n]$ as $X[f_i/b_i]$. Here is one rule for let-expressions, when *dec* declares b-variables $b_1, \ldots, b_n$:

$$\frac{\Gamma;\ dec\langle f_i/b_i\rangle \vdash (\texttt{let}\ decs\ \texttt{in}\ e\ \texttt{end})[f_i/b_i] \approx M \quad f_i \notin decs, M, e}{\Gamma \vdash \texttt{let}\ dec\texttt{;}\ decs\ \texttt{in}\ e\ \texttt{end} \approx M}$$

The rule accounts for the scope of the declaration *dec* by requiring that $f_i \notin M$: this means that *dec* cannot capture variables in M when it is moved to the context.

Local declarations are of the form:

$$\texttt{local}\ decs\ \texttt{in}\ decs'\ \texttt{end}$$

Again, *decs* declares b-variables, and associates to the right. Local declarations can appear in two places:

- They may form part of the context. In this case $decs'$ would declare f-variables and would associate to the left (as is usual for declarations in the context).
- Alternatively, they may form part of a let-expression or appear in the local part of a local declaration. In this case, $decs'$ would declare b-variables and would associate to the right.

The rules for local declarations move the declarations to and from the context, and are similar to the rules for let-expressions.

We can no longer say that the terms in a declaration are b-closed, as we did in Theorem 7 (4). Now, instead, we must say that the declaration as a whole is b-closed. For example `local` *decs* `in` $decs'$ `end` is b-closed if *decs* is b-closed and $decs'[f_1/b_1, \ldots, f_n/b_n]$ is b-closed, where *decs* declares $b_1, \ldots, b_n$.

4.2 Pattern Matching

We consider the case-expression `case` e `of` *match*, where *match* is pat_1`=>`e_1 `|` … `|` pat_n`=>`e_n. For simplicity we assume that patterns are exhaustive and non-overlapping.

Each pattern pat_i can bind an arbitrary number of b-variables, which may appear in e_i. Suppose pat_i binds variables $b_{i,1}\colon \tau_{i,1} \ldots b_{i,m_i}\colon \tau_{i,m_i}$. Considering pat_i as an expression, we know the following:

$$\forall b_{i,1}\colon \tau_{i,1} \ldots b_{i,m_i}\colon \tau_{i,m_i}.\,\texttt{case}\ pat_i\ \texttt{of}\ match \approx e_i$$

Call this term $\mathsf{CaseRule}(i, \texttt{case}\ e\ \texttt{of}\ match)$. It is b-closed by virtue of the fact that the universal quantifier binds every free b-variable which occurs in pat_i. We can thus give the following rule within Isabelle:

$$\frac{\vdash \Gamma \quad P = \mathsf{CaseRule}(i, \texttt{case}\ e\ \texttt{of}\ match) \quad \Gamma \vdash P\colon \Omega}{\Gamma \vdash P}$$

The term $\mathsf{CaseRule}(i, \texttt{case}\ e\ \texttt{of}\ match)$ is expressed as a primitive recursive function within Isabelle. It is easy to program: the least trivial step it must make is to determine the b-variables bound by pattern i.

5 Conclusions

Logics with explicit coding of binding and substitution are considerably more complex than logics which rely on meta-binding. However, they may be necessary if:

- there is no known way to code the logic using meta-binding;
- the encoding using meta-binding is unacceptable (because its correctness is difficult to establish, for example); or
- some proofs will need induction over terms.

These observations hold for all systems with explicit binding, not just McKinna-Pollack binding.

McKinna-Pollack binding has a well-developed and attractive meta-theory. Although rules in this style may look odd at first sight, they are in fact very easy to formulate—with experience they are simpler to formulate than rules for "standard" binding (although terms containing two classes of variables can on occasion be unwieldy).

In this paper we have shown that logics for programming languages may contain features which mean that they cannot reasonably be coded using meta-binding and substitution. We have further shown that McKinna-Pollack binding provides an effective way to code these logics.

6 Acknowledgements

Thanks to James McKinna for his patience in explaining his ideas to me. Thanks to Stephen Gilmore and Stuart Anderson for their patience in explaining my ideas to me. This work was partly supported by a SERC/EPSRC studentship. This document uses Paul Taylor's diagram macros.

References

[Alt93] Thorsten Altenkirch. *Constructions, Inductive Types and Strong Normalization.* PhD thesis, Department of Computer Science, The University of Edinburgh, 1993. Published as CST-106-93; also published as LFCS Technical Report ECS-LFCS-93-279.

[CF58] Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume I. North-Holland, Amsterdam, 1958.

[Chu40] Alonzo Church. A formulation of the Simple Theory of Types. *The Journal of Symbolic Logic*, 5:56–68, 1940.

[dB72] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972.

[DFH95] Jöelle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In *Typed Lambda Calculi and Applications, TLCA '95*, number

902 in Lecture Notes in Computer Science, pages 124–138, Edinburgh, 1995. Springer-Verlag.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL*. Cambridge University Press, Cambridge, 1993.

[Gor93] Andrew D. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In *HUG'93 Higher Order Logic Theorem Proving and its Applications, Vancouver*, Lecture Notes in Computer Science. Springer-Verlag, 1993.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and The Lambda-Calculus*. London Mathematical Society Student Texts. Cambridge University Press, Cambridge, 1986.

[McK] James McKinna. Typed λ-calculus formalised: Church-Rosser and Standardisation theorems. Manuscript in preparation.

[MP93] James McKinna and Robert Pollack. Pure type systems formalized. In M. Bezem and J.F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA'93*, number 664 in Lecture Notes in Computer Science, pages 289–305, Utrecht, March 1993. Springer-Verlag.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts 02142, 1990.

[Pau90] Lawrence C. Paulson. A formulation of the Simple Theory of Types (for Isabelle). In P. Martin-Löf and G. Mints, editors, *COLOG-88: International Conference on Computer Logic*, number 417 in Lecture Notes in Computer Science, pages 246–274. Springer-Verlag, 1990.

[Pau94a] Lawrence C. Paulson. *Isabelle — A Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

[Pau94b] Lawrence C. Paulson. *The Isabelle Reference Manual*. University of Cambridge Computer Laboratory, 1994.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, 1988.

[Sto88] Allen Stoughton. Substitution revisited. *Theoretical Computer Science*, 59:317–325, 1988.