uOttawa

L'Université canadienne
Canada's university

Reasoning Using Higher-Order Abstract Syntax in a Higher-Order Logic Proof Environment: Improvements to Hybrid and a Case Study

Alan J. Martin

Thesis Submitted to the Faculty of Graduate and Postdoctoral Studies
In partial fulfilment of the requirements for the degree of Doctor of Philosophy in Mathematics[1]

Department of Mathematics and Statistics
Faculty of Science
University of Ottawa

[1]The Ph.D. program is a joint program with Carleton University, administered by the Ottawa-Carleton Institute of Mathematics and Statistics

Abstract

We present a series of improvements to the Hybrid system, a formal theory implemented in Isabelle/HOL to support specifying and reasoning about formal systems using higher-order abstract syntax (HOAS). We modify Hybrid's type of terms, which is built definitionally in terms of de Bruijn indices, to exclude at the type level terms with 'dangling' indices. We strengthen the injectivity property for Hybrid's variable-binding operator, and develop rules for compositional proof of its side condition, avoiding conversion from HOAS to de Bruijn indices. We prove representational adequacy of Hybrid (with these improvements) for a λ-calculus-like subset of Isabelle/HOL syntax, at the level of set-theoretic semantics and without unfolding Hybrid's definition in terms of de Bruijn indices. In further work, we prove an induction principle that maintains some of the benefits of HOAS even for open terms. We also present a case study of the formalization in Hybrid of a small programming language, Mini-ML with mutable references, including its operational semantics and a type-safety property. This is the largest case study in Hybrid to date, and the first to formalize a language with mutable references. We compare four variants of this formalization based on the two-level approach adopted by Felty and Momigliano in other recent work on Hybrid, with various *specification logics* (SLs), including substructural logics, formalized in Isabelle/HOL and used in turn to encode judgments of the object language. We also compare these with a variant that does not use an intermediate SL layer. In the course of the case study, we explore and develop new proof techniques, particularly in connection with context invariants and induction on SL statements.

Acknowledgements

I would like to thank my advisor, Amy Felty, for her patience, encouragement, and guidance. I would also like to thank Alberto Momigliano for his support.

This work has been financially supported by the Natural Sciences and Engineering Research Council of Canada and by the University of Ottawa.

Dedication

This work is dedicated to my parents:

Barbara (Freeman) Martin & Harold Martin

Contents

List of Tables

Chapter 1

Introduction

The formalization of mathematical proof, originally pursued in search of a solid foundation for mathematics, has gained new currency with the help of computerized tools that make it feasible to conduct formal proofs in practice.

In pure mathematics, machine-checked formal proof has provided a more philosophically satisfactory alternative to computer-assisted proof based on the direct use of computer programs to check very large numbers of straightforward cases. A notable example is the proof of the four-colour theorem [3, 28], for which no proof without computer assistance is yet known.

In applied mathematics, it has facilitated the mathematical study of large formal systems, such as full programming languages. The proofs of key properties of such systems tend to involve many cases, with a few subtle details lurking among them. This can make it difficult to achieve confidence in such proofs without a method of checking them that neither tires of checking the numerous cases nor misses subtle flaws. Machine-checked formal proof offers a solution.

There have long been systems, such as first-order Zermelo-Fraenkel set theory with the axiom of choice (ZFC), that are believed to be sufficient in principle to formalize most of ordinary mathematics. However, the use of such systems for

constructing *actual* formal proofs brings new issues, as they must interact on the one hand with the computer implementation, and on the other hand with informal mathematical practice. These issues, and the approaches taken by current computer-based formalization tools, will be discussed in Chapter 2.

While current systems for formal proof include sophisticated tools for proof automation, in most cases they still require the human operator to provide *more* detail than an informal mathematical proof, leaving only straightforward steps to the automatic tools.

For certain concepts, the direct approach to formalization does not work as well as one might expect – and yet there are sometimes alternatives that provide nice solutions, while having interesting mathematical structure of their own. A notable example, and one that is central to this thesis, is the concept of variables and their scope, as found in logics and programming languages.

For example, consider first-order predicate logic. The formula $\forall x.\, \exists\, y.\; x \neq y$ is certainly satisfiable, for it is true of any set of cardinality greater than 1. The variable x is universally quantified, so we may instantiate it with any term. Yet if we mechanically replace x with y, we end up with the formula $\exists\, y.\; y \neq y$, which is clearly not satisfiable.

This problem is called *variable capture*: the quantifier $\exists\, y$ has "captured" the variable y in the instantiating term. For a sound logic, we need a more subtle notion of substitution, called *capture-avoiding substitution*. To instantiate the original formula with y, we first *rename* the existentially quantified variable y, e.g., to z, obtaining an equivalent formula $\forall x.\, \exists\, z.\; x \neq z$. (A poor choice of the replacement name can also lead to variable capture, e.g., if we replace y with x.) We may then replace x with y to obtain the correct instantiation $\exists\, z.\; y \neq z$.

In the ordinary mathematical treatment of such systems, it is common to work with terms "up to renaming of bound variables", i.e., to work with equivalence classes of terms. (See, for example, Pierce [62], section 5.3.) This may be combined with

the choice of convenient representatives for those classes, e.g., *Barendregt's variable convention* ([5], p. 26) which stipulates that bound variables in terms are always chosen to be different from all free variables present. In this way the issue of variable capture may be neatly sidestepped. Well-formedness of functions on terms is generally left implicit.

However, in formalizing the theory of logics and programming languages with the help of software *proof assistants* such as Isabelle/HOL [56] or Coq [6], we do not have the luxury of an intelligent human reader with the ability to reason informally yet soundly, so we must replace our informal conventions with something more precise. There are several possibilities:

- We could formalize the standard named-variable syntax of terms. However, since a term may be duplicated as a result of substitution, and later one instance may need to be substituted for a variable in the other, we cannot avoid the need to rename bound variables at some point.

  This may be done using a choice function that takes a list of variables or terms and produces a *fresh* variable, distinct from all the variables in its argument, otherwise arbitrary but deterministically chosen. For example, Melham [48] used this approach to formalize the π-calculus in HOL. However, he noted that much of the resulting formal theory dealt with such syntactic issues, which is undesirable if we want to go beyond syntax to nontrivial metatheory.

- We could use a canonical representation of terms up to renaming of bound variables. One such representation that is well known is *de Bruijn indices* [15]. This approach consists of replacing variable names with natural numbers, which identify a (nameless) variable-binding operator by counting outward from the variable occurrence in question. When not enough variable-binding operators are present in the expression, the index is said to be *dangling*. Dangling indices can be used as free variables, though their numbering changes under

each variable-binding operator. This representation is widely used in software implementation, but it has serious drawbacks as a human-readable notation. (We will use it in Section 3.2.2.)

A related option is to use equivalence classes of terms as in the usual informal treatment. These can be difficult to work with in formal proof systems such as Isabelle/HOL or Coq; however, this approach has been used with some success by Urban in the form of a *nominal datatype* package [73] for Isabelle/HOL.

There are also some more abstract mathematical treatments of variable binding, using certain presheaf categories [22, 34] or a somewhat unusual variant of set theory [23]. (Urban's nominal datatype package uses ideas from the latter approach.)

- Pierce ([62], Ch. 6) lists two other options. We could use *explicit substitutions* [1] (or a related mechanism) to avoid the need for a substitution operation; or we could use a system such as *combinatory logic* [14] that avoids variables entirely, by working only with *closed terms* (having no free variables) and building them from a small set of primitive *combinators* without the use of bound variables. However, variables and substitution are useful concepts, so these approaches entail significant tradeoffs; we will not consider them further here.

- We could represent the variable-binding constructs of terms using a variable-binding construct (λ-abstraction) of the formal proof system, thus representing the arguments of these constructs as *functions* in that system. This approach is called *higher-order abstract syntax* (HOAS) and dates back to Church's higher-order logic [11]. It allows us to exploit the underlying system's support for manipulating variables, which is often simpler to use than any explicit formalization of variables.

We focus on the latter option, the use of higher-order abstract syntax to specify

and reason about formal systems. The Hybrid system, developed by Ambler, Crole, and Momigliano [2], aims to support this approach in the modern proof assistants Isabelle/HOL and Coq.

Isabelle/HOL is based on an extension of higher-order logic [57], while Coq is based on a type theory called the Calculus of Inductive Constructions [6]. Both systems have a function type and λ-abstraction as required, but they also have logical constants at all types, unlike the *logical frameworks* (such as LF [32]) in which HOAS is more commonly used. This makes their function types "too large" for HOAS, in two senses:

- They contain elements with irreducible occurrences of logical constants, which thus do not represent syntax;
- The function space $\tau \Rightarrow \tau$ has a larger cardinality than τ, so a variable-binding operator represented as a functional Φ of type $(\tau \Rightarrow \tau) \Rightarrow \tau$ cannot be injective. This makes it unsuitable for syntax, for we cannot uniquely recover the argument F from a term of the form Φ(F).

Hybrid's solution to both problems is to use only a *subset* of the function type, identified by a predicate called `abstr`. It builds a type *expr* of terms with a HOAS variable-binding operator *definitionally* in terms of a de Bruijn index representation. (This is discussed further in Section 3.1.)

By providing HOAS in a modern proof assistant, Hybrid automatically gains the latter's capabilities for meta-theoretical reasoning. This approach is intended to provide advantages in flexibility and proof automation, in contrast to systems that directly implement logical frameworks, such as Twelf [60], which must build their own meta-reasoning layers from the ground up.

In contrast to established systems such as Twelf, Hybrid is currently a research prototype, and still very much a work in progress. In Chapter 3, we present a variety of improvements to the Hybrid system as implemented in Isabelle/HOL. We also prove

representational adequacy of Hybrid's HOAS for first-order named variable syntax, improving on a prior result by Crole [13].

Higher-order abstract syntax in logical frameworks [58] is generally combined with the use of *hypothetical* and *parametric judgments*. These techniques consist of the use of an implication connective and universal quantifier in the framework's specification layer to represent logical aspects of the *object language* (i.e., the system to be formalized) using the specification layer's logical contexts. These techniques have been found to combine synergistically with the use of HOAS [21].

We may consider applying these techniques in a proof assistant rather than a logical framework; however, McDowell and Miller's work [46, 47], shows that there is a tradeoff between the use of these techniques (known as "direct" or "shallow" encoding styles) and the ability to state and prove metatheoretic properties in the system where we are representing the object language. This is not a problem for logical frameworks, as they do not aim to perform meta-theoretic reasoning in the specification layer. However, it is an obstacle to the use of these techniques with Hybrid.

To successfully combine these techiques with Hybrid's HOAS and meta-theoretic reasoning, recent work using Hybrid [21, 20, 49] has adopted a *two-level* approach. This consists of formalizing a *specification logic* (SL) in the proof assistant, and then formalizing the judgments of the object language in the SL with the use of hypothetical and parametric judgments, rather than directly in the proof assistant. It is based on the methods used, e.g., by McDowell and Miller to overcome similar obstacles in logics with definitional reflection such as FO$\lambda^{\Delta\mathbb{N}}$ [46].

The advantage of using a specification logic is that we may use a more direct encoding style to represent the object language in the SL, in search of the kinds of synergies with HOAS found in logical frameworks; yet use a less direct encoding style to represent the SL in the proof assistant, to preserve the ability to perform meta-theoretic reasoning. By proving properties of the SL once and for all, and then reusing

it for many object languages, it should be possible to minimize the disadvantages of the less direct encoding of the SL in the proof assistant.

The specification logic is typically a deliberately weak logic, e.g., with implication restricted to atomic antecedents, that uses a *backchaining* rule to support logic-programming-style specification of object-language judgments [21, 46, 47]. Substructural logics such as linear logic [26] and ordered logic [65, 66] have also been used [21, 46, 52].

In Chapters 5 to 10, we present a case study of subject reduction (i.e., type safety) for Mini-ML with references ([9], see also Chapter 4), in which we develop further proof techniques for the two-level approach, and extend it to a larger object language than those previously formalized in Hybrid. We also compare various SLs (intuitionistic, linear, and ordered linear) and encoding techniques, and compare the two-level approach with the use of a less-direct encoding style (i.e., with explicit contexts, see [46, 47]) directly in Isabelle/HOL (with no SL).

1.1 Summary of Contributions

The contributions of this thesis fall into two areas: improvements to Hybrid and its metatheory, and the case study of subject reduction for Mini-ML with references in Isabelle/HOL and Hybrid. They include both traditional mathematical proof and material formalized in Isabelle/HOL.

Improvements to Hybrid

Chapter 3 presents several improvements to the Isabelle/HOL version of Ambler, Crole, and Momigliano's Hybrid system [2], and an adequacy result improving on the one proved by Crole [13].

- Starting in Section 3.2.3, we modify Hybrid's type of terms, *expr*, to better

hide its implementation in terms of de Bruijn indices, by excluding at the type level terms with dangling indices. This simplifies the representation of object languages by eliminating the need to carry a predicate for this purpose (called "proper" in [2]) along with Hybrid terms in meta-theoretic reasoning.

- In Sections 3.2.4 and 3.2.6, we modify Hybrid's HOAS variable-binding operator, LAM, to support a stronger injectivity property with only one abstr premise rather than two. This allows abstr premises of introduction rules for inductively defined predicates, as used in representing object-language judgments, to appear as conclusions, rather than premises again, in the corresponding elimination rules. That in turn reduces the need to carry abstr conditions in meta-theoretic reasoning, as they are more often available where they are needed.

  This improvement depends essentially on the previous one, for reasons that are explained in Section 3.2.6.

- In Section 3.2.9, we formally prove that a version of abstr for two-argument functions (as described by Momigliano et al. in [50]) is equivalent to a conjunction of one-argument abstr conditions on "slices" of the function (fixing one argument).

  We use this result to prove a case-distinction lemma for functions satisfying abstr, and a lemma abstr_LAM that enables compositional proof of abstr conditions at the HOAS level, without conversion to de Bruijn indices as required in [2].

  These two lemmas complete Hybrid's characterization of its type *expr* in terms of properties stated at the HOAS level, as illustrated by the fact that the proof of adequacy in Section 3.4 does not involve de Bruijn syntax. In addition to the practical benefits of the added lemmas, this makes Hybrid a more mathematically satisfactory theory.

We also eliminate the need for a special-purpose tactic for proving abstr conditions (as used in [2]) in favour of Isabelle's general-purpose **simp** and **auto** proof methods, by declaring rules for Isabelle's simplifier and classical reasoner. These consist of abstr_LAM together with some simpler properties from Section 3.2.4.

- In Section 3.2.7, we set up rewrite rules for Isabelle's simplifier to convert automatically between HOAS at type *expr* and the underlying de Bruijn index representation at a separate type *dB*, controlled by the application of type-conversion functions dB :: $expr \Rightarrow dB$ and expr :: $dB \Rightarrow expr$.

  As a result of the previous improvement, such conversion is now included only for illustrative purposes.

- In various parts of Section 3.2, especially Section 3.2.4, we generalize and simplify certain internal parts of the formal theory that constitutes Hybrid. These changes are later applied in work on generalizing abstr to functions of more than one argument, as described below.

- We replace the tactic-style proofs of the previous version of Hybrid with Isar proofs. This style of proof (which will be described briefly in Section 2.2 on Isabelle/HOL) is both more readable and more robust against changes to the underlying proof assistant (e.g., in upgrading to a more recent version).

- In Section 3.3, we define a version of abstr for a representation of n-argument functions, and prove an induction principle for such functions. This induction principle improves on Hybrid's usual form of induction, by using a representation of open terms that supports HOAS-style substitution by function application, though it does require keeping track of variable names (or numbers).

  This work is experimental, and integration into Hybrid remains as future work.

- In Section 3.4, we prove a *representational adequacy* result for (the improved version of) Hybrid, i.e., a compositional one-to-one correspondence between Hybrid's higher-order syntax and ordinary named-variable syntax for variable binding. This is done at the level of set-theoretic semantics for higher-order logic; as such, it applies directly to the actual formal theory of Hybrid, unlike Crole's result [13] which models Hybrid as a logical framework but does not prove correctness of this model.

  The proof is also based solely on the properties of *expr* formally proved by Hybrid, without reference to the underlying de Bruijn syntax. This simplifies the proof as compared with Crole's, and also illustrates Hybrid's characterization of the type *expr* as mentioned above.

In contrast to the formalized mathematics developed in Sections 3.2 and 3.3, the adequacy result of Section 3.4 is a metatheoretical result developed using traditional mathematical proof.

Case study

Chapters 5 to 10 present a case study of the formalization of subject reduction for Mini-ML with references in Isabelle/HOL and Hybrid.

- We complete the largest case study to date in Hybrid.

- We formalize operational semantics and subject reduction for mutable references in Hybrid for the first time. (This part of the formalization makes use of the substructural features of a linear specification logic.)

- In Chapter 10, we compare the two-level approach to formalizing object-language judgments (as used in Chapters 6 to 9) with the use of a less direct encoding style directly in Isabelle/HOL (in Chapter 5). We also compare the

four formalizations based on the two-level approach, which use different SLs and different encoding techniques. (The SLs themselves, and most of the encoding techniques, are not new.)

- We explore alternatives to the usual method of induction on SL statements (from [46]), which uses a natural-number argument as an induction measure. One alternative (in Section 7.4) is the use of structural induction on the inductively-defined SL sequent predicate. Another (in Chapter 8) is the use of a possibly-infinite *ordinal* induction measure, together with transfinite induction. The latter approach, in particular, allows inductive proofs of nearly identical structure to that obtained with the use of a natural-number argument, while eliminating the need to calculate bounds on derivation height when using infinitely-branching rules (such as all_i in Chapter 8).

- In Section 8.3, we explore the use of Isabelle's **locale** mechanism for modular definition of the *context invariants* needed to reason about SL statements in the proof of subject reduction.

While the case study consists of a somewhat more technical variety of formal proof as compared with Hybrid, it does contain points of mathematical interest such as the use of transfinite induction in Chapter 8.

1.2 Outline of the Thesis

Chapter 2 provides background on formal proof in general, and describes the Isabelle/HOL system [56] in which the formal theories described in this thesis were constructed.

Chapter 3 presents Hybrid, an Isabelle/HOL theory that provides higher-order abstract syntax (HOAS). The version presented there improves on [2]; this chapter thus serves both as background for the case study to follow, and to present contri-

butions of its own. Some experimental work not yet integrated into Hybrid is also presented. Representational adequacy of Hybrid is proved in Section 3.4.

Chapter 4 specifies a subset of the programming language ML, which extends Mini-ML with mutable references, in the form that will be used as the object language (OL) for the case study of Hybrid and HOAS in Isabelle/HOL to follow. It defines the syntax, continuation-style operational semantics, and type system, and states the type-safety theorem (a.k.a. subject reduction) that is the focus of the case study's formalizations. (This object language is based on [9], with minor modifications and with many omitted cases filled in.)

Chapters 5 to 9 present a case study in Isabelle/HOL and Hybrid consisting of five successive formalizations of Mini-ML with references and its subject reduction theorem. The first uses a "one-level" approach with explicit typing contexts; the second uses a "two-level" approach where the judgments of the OL are encoded in a *specification logic* (SL); and the remaining three extend the two-level approach by adding sub-structural logic features to the specification logic. Chapter 10 makes some concluding observations regarding the case study, and discusses related work.

Chapter 11 concludes with a summary of results and a discussion of future work.

1.3 Higher-Order Abstract Syntax

We introduce higher-order abstract syntax by way of an example; for a more comprehensive presentation, see [32, 59].

Higher-order abstract syntax (HOAS) is a technique for representing the syntax of object languages (OLs) in a meta-language based on a typed λ-calculus. For an example, we consider as an object language the *untyped* λ-calculus:

$$e ::= x \mid (e_1\ e_2) \mid (\lambda x.\, e)$$

where e, e_1, and e_2 represent expressions and x represents a variable, and we assume

a countably infinite set of variables.

A HOAS representation of this object language consists of a type *exp* to represent expressions, and constants

$$\mathsf{APP} \quad \text{of type} \quad exp \to exp \to exp$$
$$\mathsf{LAM} \quad \text{of type} \quad (exp \to exp) \to exp$$

to represent OL application and λ-abstraction respectively. APP is simply a binary operator on the type *exp*, written in curried form; this is a first-order construct, which could equally well appear as a constant in a first-order theory, or as a constructor of an inductive datatype in a system such as Isabelle/HOL or Coq. LAM, on the other hand, is a second-order construct, specifically an operator taking a function-type argument, which is the HOAS way of representing a variable-binding operator.

We translate OL expressions to terms of type *exp* by a function ϕ where

$$\phi(x) = x^*$$
$$\phi(e_1\ e_2) = \mathsf{APP}\ \phi(e_1)\ \phi(e_2)$$
$$\phi(\lambda x.\, e) = \mathsf{LAM}\ (\lambda x^*.\, \phi(e))$$

where the function $(\_)^*$ is a bijective mapping from OL variables to variables of type *exp*. This translation preserves α-equivalence and capture-avoiding substitution:

$$e_1 \sim_\alpha e_2 \quad \text{iff} \quad \phi(e_1) \sim_\alpha \phi(e_2)$$
$$\phi\big(e_1[e_2/x]\big) = \phi(e_1)\big[\phi(e_2)/x^*\big]$$

Thus, properties of the meta-language concerning α-equivalence and substitution transfer directly to the object language. In the case where the meta-language includes a notion of equality up to $\alpha\beta$-equivalence, this allows us to treat variables using something close to Barendregt's convention and perform OL substitution using meta-language function application.

A first-order representation would differ from the HOAS representation above by having a separate type of variables *var*, with an additional constant VAR of type

$var \to exp$ and with LAM taking two arguments, a variable and an expression. With such a representation, we would have to formalize α-equivalence and substitution explicitly, and prove their basic properties, as in [48].

Up to this point, we have only required that the meta-language have function types, application, and λ-abstraction. However, to prove properties of the OL in terms of its HOAS representation, we need *representational adequacy*: every term of type *exp* should be equivalent in an appropriate sense[1] to $\phi(e)$ for some OL expression e. This causes problems when attempting to apply HOAS in an expressive meta-language such as Isabelle/HOL or Coq, as we will see in Chapter 3.

[1] In a logical framework, the appropriate equivalence is definitional equality (and we might have to exclude non-fully-applied terms); in our adequacy proof (Section 3.4), it is semantic equality in a set model.

Chapter 2

Formal Proof

A large portion of this thesis concerns formal proofs in Isabelle/HOL. The subject of formal proof is a topic of considerable recent interest in mathematics [44], and one that leads to some philosophical questions regarding the nature of mathematical proof (see, for instance, [43, ch. 4]). In this chapter we will briefly explore this topic, and also describe some aspects of Isabelle/HOL relevant to this work.

2.1 Motivation and Examples

The development of formal systems for mathematics long predates the invention of computers. In the early part of the 20^{th} century, the main interest in these systems (notably Whitehead and Russell's *Principia Mathematica* [77]) was as a foundation for mathematics, as set out in Hilbert's program (see [78]). Gödel's incompleteness theorems showed that the most ambitious goals of this program, notably a finitary consistency proof for the methods of ordinary mathematics, were unachievable. However, systems such as Zermelo-Fraenkel set theory with the axiom of choice (abbreviated ZFC) are still generally believed to be capable of formalizing

at least most of ordinary mathematics[1]. This is *formalization in principle*: without computers, actually carrying out the encoding of large portions of mathematics in such systems would be tedious and of dubious value.

The development of computers has provided the means and motivation for *formalization in practice*. Systems such as Isabelle/HOL [56] and Coq [6], while generally not yet able to automatically prove nontrivial results, have reduced the burden of tedious detail and "bookkeeping" to the point where it is feasible to formalize existing mathematical proofs. There are a variety of reasons to pursue such efforts, and a corresponding variety in the nature of the formal proofs.

- At one extreme, there are results that have been obtained by computer-assisted methods that would have been infeasible to carry out on paper. The leading example is Haken and Appel's proof of the four-colour theorem [3, 4]. In this proof, the problem was reduced by ordinary mathematical reasoning to a certain property of the objects in a large finite set, and a purpose-built computer program was used to check the billions of cases thus obtained. However, the fallible nature of computer programs (and programmers) makes this approach less satisfactory than a typical mathematical proof. The traditional part of the proof was also very large (hundreds of pages), making it difficult to check in detail. For these reasons, the result was at first not universally accepted (see [43, pp. 138–139]).

  Gonthier [27, 28] improved on this situation by constructing a formal proof of the four-colour theorem in the Coq system [6], a modern interactive *proof assistant* based on a constructive type theory called the Calculus of Inductive Constructions (CiC). Coq's support for *computational reflection*, in which proofs may include computations performed by normalization of terms in the type theory, allowed Gonthier to follow essentially the same proof strategy as Haken

[1] An important exception is the metatheory of ZFC itself.

and Appel (as simplified by Robertson et al. [69]), using the computer to check billions of cases, but with the correctness of this procedure guaranteed by Coq's type system. This approach also turned out to be useful for parts of the proof that had originally been done in the traditional mathematical way without computer assistance [27].

- At the other extreme, there is *proof-carrying code* [54], which makes use of formal proofs generated automatically, e.g., by a compiler, to express safety properties for software. These proofs may incorporate lemmas that formalize mathematical results, but they consist mostly of a formalization of information obtained by compiler techniques such as static analysis and type-checking. Such applications are arguably a part of applied mathematics, but they are usually pursued as computer science.

  Formal proof is also used for *correctness* properties of software, which are more complicated and may require considerable human effort, sometimes including nontrivial mathematics. (Gonthier's use of computational reflection [27] involved such correctness proofs.)

- Another area of mathematical research is the use of formal proof to obtain increased confidence in the correctness of proofs that can be (and may have been) carried out on paper, but that contain many tedious cases, making verification of the proof by other mathematicians a difficult task.

  Such proofs often occur in the mathematical study of properties of programming languages, which is part of the interdisciplinary subject of theoretical computer science and has been pursued by both mathematicians and computer scientists. Research is often conducted on highly simplified programming languages to keep the numbers of cases manageable, but it is also desirable to be able to prove properties of more realistic programming languages. This is one of

the goals of Hybrid, a system for which improvements are developed in the present work (Chapter 3). The development and theory of such systems, using techniques such as HOAS, tends toward the mathematical side of the subject; however, the formal proofs in the subsequent case study (Chapters 5 to 9) also have considerable mathematical content, e.g., the use of transfinite induction in Chapter 8.

If we are to use formal proof to obtain increased confidence in mathematical results, then we must ensure that the software tools used to conduct such proofs correctly implement the intended formal systems. Existing systems use several approaches:

- HOL [35] and Isabelle/HOL [56] follow the *LCF approach* due to Milner (see [31]), in which proofs are represented as objects of an abstract data type (traditionally called *thm*) in a type-safe programming language (a variant of Milner's ML). The code implementing this type has the sole task of *checking* proofs, and is kept small and simple so that it can be carefully verified by hand. The type-safety guarantees of the programming language ensure that errors in the rest of the system cannot endanger the soundness of proofs represented in this way. (Indeed, the type *thm* need not actually store a representation of the proof, although it often does for reasons described below.)

- Coq [6] and several other systems are based on constructive dependent type theory and the judgments-as-types approach, where formal statements are types and proofs are objects inhabiting those types. In this case it is the type-checking code that must be verified to ensure that it implements the intended type system. This code is again kept small and simple to facilitate verification.

- Some systems, including both Coq and Isabelle/HOL, provide *proof terms* that can be verified by independently constructed proof-checking software. Such

software can be constructed by many people using a variety of hardware, operating systems, and programming languages, so that an unlikely coincidence of flaws would be required for an incorrect proof to be accepted.

- Many other approaches have been used, e.g., translation from a more complicated formal system into a simpler one.

Soundness and consistency properties of the formal system itself are also important (and have failed at times, e.g., as mentioned in [75]). As a consequence of Gödel's incompleteness theorems, we cannot expect to establish such properties in an absolute way; however, we can prove them informally (or formally) in a well-established metalanguage with sufficient consistency strength, which in practice rarely needs to go much beyond ZFC set theory.

Correctness of the layers of hardware and software underlying the proof-checking program is also a difficult issue. In practical terms, diligent application of the methods described above is sufficient to eliminate any realistic possibility of a false proof being accepted. Philosophically, however, these methods are not entirely satisfactory. We do not yet have general-purpose computers and operating systems that are *proven* correct; and any such proof would on the one hand depend on modelling the *physical* behaviour of the machine, which is not directly susceptible to mathematical proof, and on the other hand would be so large and complicated as to likely require computer assistance – a circularity that might be difficult to resolve! These issues are discussed in detail by Pollack [68]; see also [43, Ch. 8].

2.2 The Isabelle/HOL Proof Assistant

Isabelle/HOL [56] is an interactive proof assistant that implements an extension of Church's higher-order logic [11], and is built within the generic framework of Isabelle [40]. It is based [57] on Gordon's HOL system [30].

We briefly describe the notation we will use to present definitions and statements formally proved in Isabelle/HOL, along with some features of the system that we will need. Many of these features are more recent than Nipkow et al.'s 2002 book [56]; current information may be found in the Documentation section of the Isabelle project website [40]. (We will define set-theoretic semantics for Isabelle/HOL later, in Section 3.4.1.)

Our formal theory development is based on Isabelle/HOL 2008, but we do not anticipate major difficulties in updating our work for newer versions of Isabelle/HOL. (An archived version of the Isabelle website, with the documentation and software of Isabelle 2008, is available at [41].)

Notation

We use a typeset version of Isabelle/HOL's concrete syntax [57] that is similar to the default output of Isabelle's document-preparation system.

We distinguish the syntactic classes of Isabelle/HOL by typefaces: roman type for variables and lemma/theorem names, sans serif type for defined constants, italic type for types, and boldface type for Isabelle keywords.

Isabelle/HOL is built on top of Isabelle/Pure, which is a logical framework with only a few connectives: universal quantification ($\bigwedge$), implication ($\Longrightarrow$), and equality ($\equiv$). These connectives are provably equivalent to the corresponding Isabelle/HOL connectives within Isabelle/HOL, but they are often used in statements of definitions and lemmas. (We will sometimes also use a reversed implication arrow $\Longleftarrow$ when it helps readability.) The notation $[\![\mathrm{A}_1; \mathrm{A}_2; \ldots; \mathrm{A}_n]\!] \Longrightarrow \mathrm{B}$ abbreviates the nested implication

$$\mathrm{A}_1 \Longrightarrow (\mathrm{A}_2 \Longrightarrow \cdots (\mathrm{A}_n \Longrightarrow \mathrm{B}) \cdots).$$

As a logical framework, Isabelle/Pure also has λ-abstraction (λ x. B) and function application (denoted by juxtaposition). These are used directly in Isabelle/HOL,

which is a form of HOAS. Function application has the highest precedence and associates to the left.

Function types are denoted $A \Rightarrow B$, and the notation $\left[A_1, A_2, \ldots, A_n \right] \Rightarrow B$ abbreviates a type of n-ary functions in curried form,

$$A_1 \Rightarrow (A_2 \Rightarrow \cdots (A_n \Rightarrow B) \cdots).$$

We write x :: A to indicate that x has type A. This notation may also be used inside an Isabelle/HOL expression to specify a type constraint. Isabelle/HOL supports type inference, so it is usually unnecessary to give types for terms. We will likewise omit types when they are clear from context.

We write "a *type_constr*" for a type constructor *type_constr* applied to one argument a, or "$(a_1, a_2, \ldots, a_n)$ *type_constr*" when there are multiple arguments.

We use the usual logical and mathematical symbols for Isabelle/HOL connectives and quantifiers ($\longrightarrow$, $\wedge$, $\vee$, $\neg$, $\forall$, and $\exists$), equality and inequalities ($=$, $\neq$, $<$, $\leqslant$, etc.), set membership and subset ($\in$ and $\subseteq$), and function composition ($\circ$).

The *functional update* notation f (x := y) stands for the function that agrees with f on all arguments except x, for which it takes the value y.

List notation consists of a binary operator # that is used to add an element to the front of a list, in the form (h :: a) # (t :: a *list*); the empty list is denoted $\cdot$. There is also a list concatenation operator @. Membership in a list is denoted by an infix operator mem. The function set :: (a *list*) $\Rightarrow$ (a *set*) gives the set of elements in a list.

Free variables in Isabelle/HOL statements and definitions are implicitly universally quantified. A statement of the form some_predicate (λ x. a) actually means $\bigwedge$ a. some_predicate (λ x. a); since the quantifier is outside the λ-abstraction, the bound variable x may not occur in a. Thus, such a statement expresses a property of *constant* functions, not arbitrary functions. For the latter purpose, we must use a free variable of higher-order type, in the form some_predicate (λ x. A x).

However, when defining mere *notation*, this convention does not apply: a free variable will stand for an arbitrary term with possible occurrences of all bound variables in scope.

Features

We list here a number of Isabelle/HOL features that we will need.

Inductive datatypes (see [57, § 2.6])

The command **datatype** defines an inductive datatype. It is best described by example:

datatype a *btree* = Leaf a | Node (a *btree*) (a *btree*)

This command defines a type constructor *btree*, with one type parameter a, and constructors Leaf :: $a \Rightarrow (a\ btree)$ and Node :: $\left[a\ btree, a\ btree \right] \Rightarrow (a\ btree)$. It also automatically proves many properties of the type (a *btree*) and its constructors, of which three are primitive:

- Distinctness:

  $$(\mathsf{Leaf}\ \mathrm{a}) \neq (\mathsf{Node}\ \mathrm{t}_1\ \mathrm{t}_2)$$

- Injectivity:

  $$(\mathsf{Leaf}\ \mathrm{a} = \mathsf{Leaf}\ \mathrm{a}') = (\mathrm{a} = \mathrm{a}')$$
  $$(\mathsf{Node}\ \mathrm{t}_1\ \mathrm{t}_2 = \mathsf{Node}\ \mathrm{t}_1'\ \mathrm{t}_2') = (\mathrm{t}_1 = \mathrm{t}_1') \wedge (\mathrm{t}_2 = \mathrm{t}_2')$$

- Structural induction:

  $$[\![\bigwedge \mathrm{a}.\ \mathrm{P}\ (\mathsf{Leaf}\ \mathrm{a});$$
  $$\bigwedge \mathrm{t}_1\ \mathrm{t}_2.\ [\![\mathrm{P}\ \mathrm{t}_1; \mathrm{P}\ \mathrm{t}_2]\!] \Longrightarrow \mathrm{P}\ (\mathsf{Node}\ \mathrm{t}_1\ \mathrm{t}_2)]\!] \Longrightarrow \mathrm{P}\ (\mathrm{t} :: a\ btree)$$

Distinctness and injectivity are automatically added to the set of rewrite rules for Isabelle's simplifier; as a result, any equality with datatype constructors on

both sides will be simplified, and there is little need to mention these properties explicitly in proofs.

A size function and a definition-by-cases operator are automatically defined, and their basic properties proved. A case-analysis rule is derived from the structural induction rule. Functions on a datatype may be defined by primitive recursion, by declaring them with **consts** and then giving recursive equations with **primrec**.

We will reserve the term *constructor* for constants defined using **datatype**, or defined in some other way but satisfying all of the same properties, as in the case of $0 :: nat$ and $\mathsf{Suc} :: nat \Rightarrow nat$. (The command **rep_datatype** may be used to derive all the properties of a datatype from the three primitive properties listed above.)

Inductively defined predicates

A predicate may be defined using the **inductive** command, by giving a list of *introduction rules*. We give a simple example:

inductive even :: $nat \Rightarrow bool$
where zero_is_even: even 0
| next_even: even x $\Longrightarrow$ even (x + 2)

An induction rule for the predicate is automatically proved. Corresponding *elimination rules* may be generated automatically.

Some other features, such as type definitions (**typedef**), general recursive functions (**function**), axiomatic type classes (**class**), and locales (**locale**) will be explained when they are first used.

Formal Proof

Isabelle theories are specified in a language called *Isar* [74]; the name stands for *intelligible semi-automated reasoning*, and the general approach is inspired by Mizar [70] (see [76] for a comparison). Isar defines an *outer syntax* of theory and proof commands, within which quoted strings are used for Isabelle/HOL expressions and types, with their own separate and extensible *inner syntax*.

Isabelle supports two distinct styles of proof:

- In *tactic-style proof*, a goal is stated and then logical rules, lemmas, and automatic proof methods are applied in a goal-directed way using the command **apply**. Each step may solve the goal or leave one or more subgoals. Formulas and expressions are usually not stated explicitly after the initial goal; to follow the reasoning, it is typically necessary to step through the proof interactively to see the subgoals. This style of proof is useful for exploration and can lead to shorter proofs, but because intermediate results and even the tree structure of subgoals are implicit, it has significant disadvantages in readability and maintainability of proofs.

- *Isar-style proof* uses a block-structured language of proof commands. A statement is accompanied by a single proof step; either it proves the goal at once, or a block is opened in which subgoals are *stated* and proved. Intermediate results may also be stated and proved within such a block. This style of proof puts the emphasis on intermediate results rather than the rules by which they are obtained; it can be used to construct readable formal proofs approximating the style of ordinary mathematical proof.

The two styles may be freely intermixed. We use the Isar proof style almost exclusively, with occasional recourse to **apply** in situations where the Isar-style alternative is worse, such as when case analysis must be performed on a particular

variable in all cases of a proof by induction.

We generally use automatic proof methods whenever they are successful, which yields shorter proofs but not necessarily readable ones, as important steps may be hidden. There are two main automatic proof tools: the *simplifier* which operates by rewriting using proven equations, and the *classical reasoner* which operates by proof search using natural-deduction-style introduction and elimination rules. The **auto** proof method combines both techniques. We may add our own lemmas to Isabelle's *simpset* and *claset* so that they may be used by the automatic proof tools; either semi-permanently using *attributes*, or temporarily for a single automatic proof step.

Extensions

Isabelle supports a modular system of theories, in which each file begins with a **theory** command giving a name to the theory it defines and specifying one or more theories on which it depends. The theory Main of Isabelle/HOL includes many basic mathematical constructs including everything described above, but we will have occasion to use certain other theories as well.

Two of these theories are found in the `Library` directory of the Isabelle/HOL distribution, and may be loaded simply by naming them as dependencies:

- The theory Multiset defines a type (a *multiset*) of multisets with elements of type a.

  The function count :: $[\, a$ *multiset*, $a \,] \Rightarrow$ *nat* gives the number of occurrences of x in a multiset M as (count M x). Multiset membership is defined as (x $\in$ M) = (count M x $>$ 0). The singleton multiset with element x is denoted {{ x }}, and the empty multiset is denoted $\square$.

  We use an abbreviation mset for the function multiset_of :: (a *list*) $\Rightarrow$ (a *multiset*) that converts a list to a multiset by forgetting the order of elements.

- The theory Countable defines an axiomatic type class COUNTABLE with one axiom stating the existence of an injective function to the type of natural numbers.

The third is a theory Ordinal from the Archive of Formal Proofs [38], which defines a type *ordinal* of countable ordinals.

Chapter 3

Improvements to Hybrid

3.1 An Abstract View of Hybrid

This section will present an overview of the present version of Hybrid, which combines previous work (especially from [2]) and our contributions, without necessarily distinguishing the two. Details of the theory, and discussion of what parts of Hybrid are the contributions of this thesis, will follow in the subsequent sections.

Hybrid takes the form of an Isabelle/HOL theory, built around a type *expr* that serves as abstract syntax. It is intended to be used for representing object languages (OLs) such as programming languages, logics, and other formal systems. It aims to provide *higher-order abstract syntax* (HOAS) for variable-binding constructs, and to support reasoning in Isabelle/HOL about object languages so represented, including in particular proof by induction over *expr*. It is also built definitionally on the foundation of Isabelle/HOL, introducing no new axioms[1].

Isabelle/HOL already has extensive support for first-order abstract syntax, in the form of its **datatype** package. Hybrid may be viewed as an attempt to approximate

[1]Except for the *definitional extensions* used internally by Isabelle/HOL's definition mechanisms, which introduce new constants and defining axioms together in a consistency-preserving way.

a **datatype** definition that is not well-formed because of its higher-order features:

$$\begin{aligned}\textbf{datatype}\ \mathit{expr} = \mathsf{CON}\ \mathit{con} \mid \mathsf{VAR}\ \mathit{var} \mid \mathsf{APP}\ \mathit{expr}\ \mathit{expr} \quad &\text{(notation (s \$\$ t))}\\ \mid \mathsf{LAM}\ (\underline{\mathit{expr}} \Rightarrow \mathit{expr}) \quad &\text{(notation (LAM x. B))}\end{aligned}$$

where CON represents constants, from an OL-specific type *con* (typically a trivial **datatype**); VAR may be used to represent free variables, from a countably infinite type *var* (actually a synonym for *nat*); APP represents pairing, which is sufficient to encode list- or tree-structured syntax; and LAM represents variable binding in HOAS style, using the bound variable of an Isabelle/HOL λ-abstraction to represent a bound variable of the object language.[2]

It should be noted that Hybrid only approximates *one* such pseudo-datatype, not the **datatype** package with its ability to define multiple types for first-order abstract syntax. That is, Hybrid is *untyped*, so predicates rather than types must be used to distinguish different kinds of OL terms encoded into *expr*. This is a potential area for significant improvement, but it remains as future work. (Some work in that direction has been done in another version of Hybrid, based on the Coq proof assistant; see Section 3.5, Related Work.)

The problem is LAM, whose argument type includes a negative occurrence of *expr* (underlined). This is essential for HOAS, but it is not permitted in a **datatype** definition [57, § 2.6], and it will require modifications to some of the properties expected for a constructor of a datatype; we will return to this issue later.

Hybrid does provide a type *expr* with operators CON, VAR, APP, and LAM of the appropriate types. This is enough to give an example of representing terms of a simple object language, even before considering the properties required of *expr* and its operators. We use the untyped λ-calculus with its usual named-variable syntax, using capital letters for variables (V_i, $i \in \mathbb{N}$) and λ-abstraction (Λ) to avoid ambiguity.

Hybrid was originally designed as a representation of an untyped λ-calculus [2];

[2]While APP and LAM were inspired by the untyped λ-calculus, in Hybrid they are used only as syntax, without built-in notions of β-conversion, normal forms, etc.

as such, its operators can be used directly to represent the constructs of this object language. An object language term ($\Lambda\ V_1.\, \Lambda\ V_2.\, (V_1\ V_2)\ V_3$) would be represented as (LAM (λ x. LAM (λ y. (x $$ y) $$ (VAR 3)))), which can be abbreviated to (LAM x y. x $$ y $$ VAR 3). Note that the free variable V_3 is represented using Hybrid's VAR operator, while the bound variables V_1 and V_2 are represented as bound Isabelle/HOL variables (x and y) in the function-type arguments of LAM. The names of the latter variables are irrelevant, so long as they are distinct, since Isabelle/HOL handles α-conversion automatically.

However, the preferred way to represent OL terms in Hybrid is to translate each OL construct to a list formed using Hybrid's $$ operator, headed by a constant that identifies the particular OL construct. If we use c_lam :: *con* for OL lambda-abstraction and c_app :: *con* for OL function application, we would have:

c_lam $$ (LAM x. c_lam $$ (LAM y. c_app $$ (c_app $$ x $$ y) $$ VAR 3)).

This approach has more notational overhead, but it is essential for working with more complicated OLs with many constructs and syntactic classes (such as Mini-ML with references in Chapters 4 to 9). Isabelle's ability to define abbreviations and infix notations can be used to recover a reasonable concrete syntax:

fn x y. (x $ y) $ VAR 3.

(These examples have used VAR to represent free OL variables, which means giving up some of the advantages of HOAS, as discussed below. Alternatives will be discussed in Section 3.3.)

We now turn to the properties required of *expr* and its operators to function as higher-order abstract syntax. The main requirement is a meta-theoretic property, *representational adequacy*. This can take several forms, but we will use bijectivity of a set-theoretic semantics on a λ-calculus-like subset of the Isabelle/HOL terms of type *expr*, called the *syntactic terms*:

$s ::= x \mid \mathsf{CON}\ a \mid \mathsf{VAR}\ n \mid s_1\ \$\$\ s_2 \mid \mathsf{LAM}\ x.\, s$

where s (with possible subscripts) stands for a syntactic term, x for a variable of type *expr*, a for a constant of type *con*, and n for a natural-number constant.

However, open terms present a complication. Suppose we have a theory where the semantics is bijective on *closed* syntactic terms, which it maps to a set S. Then it will map *open* terms with n free variables to functions from the Cartesian power S^n to S. But there are many such functions that do not correspond to syntactic terms; for example, the function $S \to S$ corresponding to the Isabelle/HOL term

λ x. if ($\exists$ a. x = CON a) then (x $$ x) else x

of type (*expr* $\Rightarrow$ *expr*). Indeed, there are a countable infinity of syntactic terms, while the set of functions from S^n to S is uncountable for $n \geq 1$.

Thus, Hybrid must define a *subset* of the function space to be used as its representation for open syntactic terms. This is done using a predicate abstr :: ((*expr* $\Rightarrow$ *expr*) $\Rightarrow$ *bool*). The functions satisfying abstr will be those of the form (λx. s) where s is a syntactic term with (at most) one free variable x; we call these the *syntactic functions*[3]. (Syntactic terms with more than one free variable will be handled one variable at a time; the details are given as part of the proof of adequacy in Section 3.4.)

The present version of Hybrid further aims to *characterize* the type *expr*, its operators, and the predicate abstr with formal lemmas, in such a way that adequacy follows from these lemmas without unfolding the definitions.

Isabelle/HOL **datatype** definitions achieve this in the first-order case using three properties: *distinctness* of the datatype constructors, *injectivity* of each constructor, and an *induction principle*.

In the case of Hybrid, distinctness of all the operators and injectivity of the

[3]Previous work called such functions *abstractions* [2] – thus the predicate name abstr; and called functions not satisfying abstr *exotic terms* [2, 16], because they are foreign to the intentionally weak logics typically used as specification layers for logical frameworks.

first-order operators (i.e., all except LAM) are straightforward to achieve, e.g.:

$$\forall\ (\mathrm{c} :: \mathit{con})\ (\mathrm{S} :: \mathit{expr} \Rightarrow \mathit{expr}).\ \mathsf{CON}\ \mathrm{c} \neq \mathsf{LAM}\ \mathrm{S}$$
$$\forall\ (\mathrm{s}\ \mathrm{t}\ \mathrm{s}'\ \mathrm{t}' :: \mathit{expr}).\ (\mathrm{s}\ \$\$\ \mathrm{t} = \mathrm{s}'\ \$\$\ \mathrm{t}') \longrightarrow (\mathrm{s} = \mathrm{s}') \wedge (\mathrm{t} = \mathrm{t}').$$

(These properties are used as rewrite rules for Isabelle's simplifier, to reduce equalities of Hybrid terms with known operators on both sides; typically this results in equalities where one side is just an Isabelle/HOL variable, which can then be eliminated by substitution.)[4]

Injectivity of LAM must be restricted to functions satisfying abstr; indeed, it can be proven in Isabelle/HOL that no injective function from $(\mathit{expr} \Rightarrow \mathit{expr})$ to *expr* exists, by formalizing Cantor's diagonal argument. However, the present version of Hybrid proves an injectivity property that requires an abstr condition for only one side of the equality:

$$[\![\, \mathsf{abstr}\ \mathrm{S} \vee \mathsf{abstr}\ \mathrm{T};\ \mathsf{LAM}\ \mathrm{S} = \mathsf{LAM}\ \mathrm{T} \,]\!] \Longrightarrow \mathrm{S} = \mathrm{T}.$$

This reduces the need for explicit abstr conditions in object-language encodings, because they can be transported across equalities of LAM terms. It is achieved by adding to the type *expr* an additional constant ERR, and defining LAM to take the value ERR on functions not satisfying abstr. (The constant ERR will sometimes appear as an additional case alongside the operators of Hybrid, in lemmas that impose an abstr condition for the LAM case. We also include it among the syntactic terms.)

Since abstr appears as a premise of injectivity – and it would in any case be needed to state properties of open syntactic terms – we must also include properties sufficient to characterize it. While Hybrid proves a number of lemmas regarding abstr for convenience and proof automation, the desired characterization can be given in a

[4]Indeed, most use of Hybrid's lemmas in object-language work is automated using Isabelle's simplifier and classical reasoner, and as a result, direct references to Hybrid's lemmas may be rare.

single statement:

$$\begin{aligned}
\mathsf{abstr}\ \mathrm{Y} \equiv\ & (\mathrm{Y} = (\lambda\ \mathrm{x}.\ \mathrm{x})) \\
\vee\ & (\exists\ \mathrm{a}.\ \mathrm{Y} = (\lambda\ \mathrm{x}.\ \mathsf{CON}\ \mathrm{a})) \\
\vee\ & (\exists\ \mathrm{n}.\ \mathrm{Y} = (\lambda\ \mathrm{x}.\ \mathsf{VAR}\ \mathrm{n})) \\
\vee\ & (\exists\ \mathrm{S}\ \mathrm{T}.\ \mathrm{Y} = (\lambda\ \mathrm{x}.\ \mathrm{S}\ \mathrm{x}\ \$\$\ \mathrm{T}\ \mathrm{x}) \wedge \mathsf{abstr}\ \mathrm{S} \wedge \mathsf{abstr}\ \mathrm{T}) \\
\vee\ & (\exists\ \mathrm{W}.\ \mathrm{Y} = (\lambda\ \mathrm{x}.\ \mathsf{LAM}\ \mathrm{y}.\ \mathrm{W}\ \mathrm{x}\ \mathrm{y}) \wedge \underline{\mathsf{abstr}\ \mathrm{W}}) \\
\vee\ & (\mathrm{Y} = (\lambda\ \mathrm{x}.\ \mathsf{ERR}))
\end{aligned}$$

Once again the LAM case complicates matters: the underlined occurrence of (abstr W) applies abstr to a function $\mathrm{W} :: \big([expr,\ expr] \Rightarrow expr\big)$. This should be possible by using type classes to give a polymorphic definition for abstr, but that is future work. The present version of Hybrid instead replaces (abstr W) with

$$(\forall\ \mathrm{y}.\ \mathsf{abstr}\ (\lambda\ \mathrm{x}.\ \mathrm{W}\ \mathrm{x}\ \mathrm{y})) \wedge (\forall\ \mathrm{x}.\ \mathsf{abstr}\ (\lambda\ \mathrm{y}.\ \mathrm{W}\ \mathrm{x}\ \mathrm{y})).$$

(This requires a partial formalization of the reasoning used to handle one variable at a time in the proof of adequacy.)

As for induction, it can take several forms. In the first-order case, Isabelle/HOL's datatype package provides two forms of induction:

- A *structural induction* rule.
- A well-founded *induction measure*, in the form of a function size :: $(a \Rightarrow nat)$ defined for any[5] datatype a, together with equations to simplify size applied to the constructors; and an *exhaustiveness* property for the constructors.

In the case of Hybrid, it is straightforward enough to define a size function on syntactic terms, which can be used as an induction measure. Hybrid does this for the type *expr*, which represents closed syntactic terms, and there is a straightforward extension to open terms. The proof of adequacy in Section 3.4 uses this induction

[5]In fact the size function is not defined for datatypes with infinitely-branching recursion, e.g., Definition 6.1.

measure, together with exhaustiveness properties for the types *expr* and $(expr \Rightarrow expr)$.

However, this approach is not entirely satisfactory for reasoning within Isabelle/HOL. To work with *open* syntactic terms (which may occur as subterms even if we start with a closed term), we must λ-bind their free variables; this is semantically trivial but *changes the type* from *expr* to a functional type $\big([expr]^n \Rightarrow expr\big)$, where n is the number of free variables.[6] Isabelle/HOL is thus unable to formalize the form of induction used in the proof of adequacy, because it lacks explicit quantification over types.

To work around this limitation, we must bring the induction cases back to the same type as the original object. The original version of Hybrid [2] addressed this issue by including another representation of free variables within the type *expr*, specifically named-variable syntax in the form of VAR (as illustrated above in the example of encoding OL terms). It then proved a structural induction principle for *expr* in terms of this representation, while still using the HOAS representation for bound variables. The first-order induction cases are standard, while the induction case for LAM is:

$$\forall\ \mathrm{S} :: (expr \Rightarrow expr).\quad \mathsf{abstr}\ \mathrm{S} \wedge \big(\forall\ \mathrm{n}.\ \mathrm{P}\ (\mathrm{S}\ (\mathsf{VAR}\ \mathrm{n}))\big) \longrightarrow \mathrm{P}\ (\mathsf{LAM}\ \mathrm{x}.\ \mathrm{S}\ \mathrm{x}).$$

However, falling back to named (or numbered) variables for inductive proofs means giving up some of the advantages of HOAS. To make meaningful use of such an induction principle, it is necessary to prove many technical lemmas about VAR involving freshness, substitution, etc.

The present version of Hybrid retains VAR and this induction principle, but omits the technical lemmas. As an alternative, in Section 3.3 we prove an induction principle for a type that represents n-ary functions on the type *expr*. This approach preserves the HOAS feature of substitution by function application, though it is currently experimental and integration into Hybrid remains as future work.

[6] The meta-notation $\big([expr]^n \Rightarrow expr\big)$ stands for the Isabelle/HOL type of curried n-ary functions with argument and result types *expr*. (When $n = 0$, this is just *expr*.)

(The lack of a good induction principle for syntactic terms was not a major obstacle in the case study presented in Chapters 5 to 10, because other forms of induction were available and open terms could be handled using HOAS techniques. Size induction on *expr* is nonetheless used in Lemma 7.31, which deals with a first-order part of an OL encoding where the LAM case cannot occur.)

In addition to proof by induction, it is convenient to be able to recursively define functions and predicates on syntactic terms. For datatypes, Isabelle/HOL provides primitive recursion in the form of a **primrec** command; extending this to Hybrid would require modifying the datatype package's program code. However, Isabelle/HOL 2007 introduced a new **function** command that supports general recursive definition using a well-founded termination measure; it works unmodified with Hybrid, though it faces the same type issues described above for induction.

Finally, Hybrid aims to build *expr* and its operators definitionally in Isabelle/HOL. While the description above is an informal but reasonably complete specification of Hybrid, it is not directly usable as a definition because it is circular: the arguments of LAM and abstr may themselves contain LAM, and injectivity of LAM depends on abstr. It could be formalized as an axiomatic theory, leaving consistency as a meta-theoretical problem; but instead, Hybrid is built definitionally in terms of a *first-order* representation of variable binding based on de Bruijn indices. The definitions and lemmas involved in achieving this are the subject of the next section.

3.2 Definition of Hybrid

This section presents the definitions and many of the lemmas of Hybrid. Some technical lemmas are omitted, as are most proofs, many of which use Isabelle/HOL's automatic proof methods. Informal translations of Isar proofs are given for some key lemmas, typically showing more steps than were actually needed in the formal proof to improve readability. (The Isabelle/HOL 2008 theory file for the present version of

Hybrid is available online [45].)

The present version of Hybrid is based on Hybrid as developed by Ambler et al. [2]; the theory files for that version were provided by Alberto Momigliano, and are now available online [18] (as "the original Hybrid infrastructure" for Isabelle/HOL 2002 and 2003). The discussion accompanying the theory development will specify which features are from that work and which ones are original, and explain the ways in which we improve on the original version of Hybrid.

Some of our contributions were presented in [51], and the Isabelle theory file corresponding to that paper is also available online [18] (as "Hybrid 0.2", in versions for Isabelle/HOL 2005, 2007, and 2008). The most significant change since that version is the addition of the material in Section 3.2.9, notably Lemma 3.38 (abstr_LAM). This change enabled the proof of Theorem 3.56 (Adequacy) without unfolding Hybrid's definitions, an important objective as described in Sections 3.1 and 3.4.4.

One improvement that will not be apparent, due to the omission of proofs, is that all of the theory's lemmas are proved in Isar style, rather than as tactic scripts. ("Improper" Isar constructs, emulating tactics, are used in a few exceptional cases.) This makes the formal proofs more readable, and better approximates informal mathematical practice. The Isar proofs were also quite robust when upgrading from Isabelle 2005 to Isabelle 2007 and then Isabelle 2008; the required changes were few and obvious, which is typically not the case for large theories based on tactic scripts (as noted, e.g., in [18]).

3.2.1 Outline

We begin in Section 3.2.2 by defining a datatype *dB* for first-order abstract syntax with variables represented by de Bruijn indices. In Section 3.2.3, we then define the type *expr* and its first-order operators (i.e., all except LAM) in terms of *dB* and its constructors.

In Section 3.2.4, we define the remaining operator LAM and the predicate abstr, and give some of their basic properties as lemmas. These definitions are illustrated by an example in Section 3.2.5.

In Section 3.2.6, we prove an injectivity property for LAM on arguments satisfying abstr, and state distinctness and injectivity for all of Hybrid's operators (the remaining cases being straightforward).

In Section 3.2.7, we define an inverse for the auxiliary function used to define LAM in terms of de Bruijn indices, and use it to support conversion of de Bruijn syntax to HOAS. This is used to prove exhaustiveness of Hybrid's operators for the type *expr* in Section 3.2.8, where we also define an induction measure (as a function size :: $(expr \Rightarrow nat)$) and give a structural induction principle for *expr* using VAR for free variables.

In Section 3.2.9, we generalize abstr to binary functions, and present a lemma reducing this generalization to the original predicate abstr for unary functions. This is used to complete the characterization of abstr described in Section 3.1.

3.2.2 De Bruijn syntax

The Hybrid theory defines the type *expr* in terms of an Isabelle/HOL datatype *dB*, which represents abstract syntax using a nameless first-order representation of bound variables called *de Bruijn indices* [15].

This approach differs from the original version of Hybrid [2], which used a datatype corresponding to our *dB* directly as *expr*; the significance of this difference will be explained in Sections 3.2.3 and 3.2.6. However, the datatype itself is very similar, and this section follows [2] closely.

Definition 3.1

types
$var = nat$
$bnd = nat$
datatype $a\ dB =$
$\mathsf{CON}'\ a \mid \mathsf{VAR}'\ var \mid \mathsf{APP}'\ (a\ dB)\ (a\ dB) \quad (notation\ (\text{s}\ \$\$'\ \text{t}))$
$\mid \mathsf{ERR}' \mid \mathsf{BND}'\ bnd \mid \mathsf{ABS}'\ (a\ dB)$

The constructors CON′, VAR′, and APP′ correspond to the operators CON, VAR, and APP on type *expr*, which were discussed in Section 3.1 and will be defined later. The one significant difference is that the argument of CON′ is a type parameter a, rather than a particular type *con*. This will actually be true for CON as well, and it allows Hybrid to be defined as an OL-independent Isabelle/HOL theory, and later used with OL-specific constants. (We will frequently omit this type parameter, except where it occurs in formal definitions or it is instantiated.)

The other three constructors (ERR′, BND′, and ABS′) will all be used in the definition of LAM. The constant ERR′ will be a placeholder for LAM applied to a non-syntactic function; it was not present in [2], and its significance will be explained later. The constructor ABS′ functions as a nameless binder, while (BND′ i) represents the variable implicitly bound by the $(\text{i} + 1)^{\text{th}}$ enclosing ABS′ node. If there are not enough enclosing ABS′ nodes, then it is called a *dangling index*.

As an example, consider the term

$$\underline{\mathsf{ABS}'}\ (\mathsf{ABS}'\ (\mathsf{BND}'\ 2\ \$\$'\ \underline{\mathsf{BND}'\ 1}\ \$\$'\ \mathsf{BND}'\ 0)\ \$\$'\ \underline{\mathsf{BND}'\ 0}).$$

The underlined occurrences of (BND′ 1) and (BND′ 0) both refer to the variable bound by the outer ABS′ (also underlined), while the other occurrence of (BND′ 0) refers to the variable bound by the inner ABS′. (BND′ 2) is a dangling index, because there are only 2 enclosing ABS′ nodes.

To keep track of dangling indices, Hybrid defines a predicate level :: $\left[\, bnd,\ dB \,\right] \Rightarrow bool$ such that (level i t) is true if enclosing the term t in i or more ABS′ nodes would

result in a term without dangling indices.

Definition 3.2

fun level :: $\left[bnd,\ a\ dB \right] \Rightarrow bool$
level i (CON′ a) = True
level i (VAR′ n) = True
level i (s $$′ t) = (level i s ∧ level i t)
level i ERR′ = True
level i (BND′ j) = (j < i)
level i (ABS′ s) = level (i + 1) s

A term with no dangling indices is called *proper*, and we may define an abbreviation (proper t) = (level 0 t). (These notions are standard for abstract syntax based on de Bruijn indices [2].)

Lemma 3.3

level_le_level: $[\![\text{level i s; i} \leqslant \text{j}]\!] \Longrightarrow \text{level j s}$

The lemma level_le_level shows that level is upward closed in its first argument: a term satisfying (level i) also satisfies (level j) for any j > i.

Dangling indices are the analog of free variables for de Bruijn syntax: the subterms of a proper term are not necessarily proper, but may instead be open terms represented using dangling indices. Hybrid uses this de Bruijn representation of open terms internally in defining LAM and abstr, but it is not intended to be part of *expr*; indeed, its presence there would conflict with its internal use in the definition of LAM.

3.2.3 The type "expr" of proper de Bruijn terms

In the original version of Hybrid [2], where *expr* was a datatype similar to our *dB*, many important properties were proved with proper premises. This meant that OL encodings had to build in proper conditions, and proofs of OL metatheorems had to

manipulate them; a special-purpose tactic called proper_tac was provided for this purpose.

The present work improves on that situation by using Isabelle/HOL's **typedef** mechanism to define *expr* as a bijective image of the set of proper terms of type *dB*. That eliminates the proper conditions in object-language work using Hybrid, at the expense of having to convert terms between *expr* and *dB* in defining LAM and abstr. This is a good trade-off, because those definitions are internal to Hybrid and need only be made once. It also turns out to be essential for strengthening the quasi-injectivity property of LAM, as described in Section 3.2.6.

Definition 3.4

typedef (**open**) *a expr* = {x :: *a dB*. level 0 x} **morphisms** dB expr

This **typedef** statement first demands a proof that the specified set is nonempty (which is trivial here). Then it introduces the type *expr*, the functions dB :: (*expr* ⇒ *dB*) and expr :: (*dB* ⇒ *expr*), and axioms stating that they are inverse bijections between the type *expr* and the set {x :: *dB*. level 0 x}. (Some obvious but useful consequences of these axioms, including injectivity and surjectivity of the functions, are proved automatically.) Although axioms are used, the overall mechanism is a form of definitional extension and preserves consistency of the theory.

The **open** modifier instructs Isabelle not to define a new predicate for membership in the set, which would be redundant here, since (level 0) is already such a predicate. The type *expr* has the same type parameter *a* that *dB* does, and we will frequently omit it for *expr* as well.

We may now define all of the first-order operators of Hybrid (i.e., all except LAM, with its functional-type argument) in the obvious way.

Definition 3.5

CON :: *a* ⇒ *a expr*
CON a ≡ expr (CON′ a)

$\mathsf{VAR} :: \mathit{var} \Rightarrow \mathit{a\ expr}$
$\quad \mathsf{VAR}\ \mathrm{n} \equiv \mathsf{expr}\ (\mathsf{VAR}'\ \mathrm{n})$
$\mathsf{APP} :: [\, \mathit{a\ expr}, \mathit{a\ expr}\,] \Rightarrow \mathit{a\ expr} \qquad (\mathit{notation}\ (\mathrm{s}\ \$\$\ \mathrm{t}))$
$\quad \mathrm{s}\ \$\$\ \mathrm{t} \equiv \mathsf{expr}\ (\mathsf{dB}\ \mathrm{s}\ \$\$'\ \mathsf{dB}\ \mathrm{t})$
$\mathsf{ERR} :: \mathit{a\ expr}$
$\quad \mathsf{ERR} \equiv \mathsf{expr}\ \mathsf{ERR}'$

ERR is defined as if it were a separate operator, and it will sometimes be treated as such, but it will also be generated by LAM applied to a non-syntactic function.

The functions dB and expr translate these operators to the corresponding constructors of *dB* (Definition 3.1) and vice versa.

Lemma 3.6

$\mathsf{dB}\ (\mathsf{CON}\ \mathrm{a}) = \mathsf{CON}'\ \mathrm{a}$
$\mathsf{dB}\ (\mathsf{VAR}\ \mathrm{n}) = \mathsf{VAR}'\ \mathrm{n}$
$\mathsf{dB}\ (\mathrm{s}\ \$\$\ \mathrm{t}) = \mathsf{dB}\ \mathrm{s}\ \$\$'\ \mathsf{dB}\ \mathrm{t}$
$\mathsf{dB}\ \mathsf{ERR} = \mathsf{ERR}'$

Lemma 3.7

$\mathsf{expr}\ (\mathsf{CON}'\ \mathrm{a}) = \mathsf{CON}\ \mathrm{a}$
$\mathsf{expr}\ (\mathsf{VAR}'\ \mathrm{n}) = \mathsf{VAR}\ \mathrm{n}$
$[\![\, \mathsf{level}\ 0\ \mathrm{s};\ \mathsf{level}\ 0\ \mathrm{t}\,]\!] \Longrightarrow \mathsf{expr}\ (\mathrm{s}\ \$\$'\ \mathrm{t}) = \mathsf{expr}\ \mathrm{s}\ \$\$\ \mathsf{expr}\ \mathrm{t}$
$\mathsf{expr}\ \mathsf{ERR}' = \mathsf{ERR}$

These properties are all straightforward consequences of the definitions.

Distinctness and injectivity for these operators follow from the corresponding properties of *dB* using these lemmas; however, we will defer stating those properties until Section 3.2.6, where we can include similar properties for LAM.

The (level 0) premises of the third property in Lemma 3.7 are needed because the **typedef**-generated function expr is undefined on terms with dangling indices. These premises could be eliminated by defining a more tightly-specified version of expr, satisfying the same **typedef**-generated axioms while preserving the structure of its argument except for any dangling indices. This was done in the previous version of

Hybrid [18, 51] (with the help of an auxiliary function called trim), but with systematic use of the predicate level as described below, it was found to be unnecessary.

It would be convenient to be able to make a type definition similar to Definition 3.4 for the predicate (level i) for arbitrary i :: *nat*. However, the *nat* parameter would make this a dependent type, which is not supported by Isabelle/HOL.

Nonetheless, in the present work, Hybrid's use of the predicate level is structured as if by translation from a notional extension of Isabelle/HOL with dependent types. This determines a systematic way of using level that avoids nontrivial proof obligations. (For an example of an actual translation from a language with dependent types into one without them, see [17].)

The notional type (*level* i) is replaced with the actual type *dB*, inserting (level i) subformulas to relativize quantifiers:

$$\begin{array}{lll} \forall\ \mathrm{x} :: (\mathit{level}\ \mathrm{i}).\ \mathrm{P\ x} & \text{becomes} & \forall\ \mathrm{x} :: \mathit{dB}.\ \mathsf{level}\ \mathrm{i} \longrightarrow \mathrm{P\ x}\ ; \\ \exists\ \mathrm{x} :: (\mathit{level}\ \mathrm{i}).\ \mathrm{P\ x} & \text{becomes} & \exists\ \mathrm{x} :: \mathit{dB}.\ \mathsf{level}\ \mathrm{i} \wedge \mathrm{P\ x}\ . \end{array}$$

We also insert (level i) premises to relativize the implicit universal quantification of free variables. (The lemmas proved automatically by Isabelle/HOL for the **typedef** in Definition 3.4 fit this pattern, with i = 0.)

Type-checking is simulated using simplifier rules for level, one for each construct of type *dB*. The defining equations from Definition 3.2 provide such rules for the constructors of *dB*. However, they indicate the need for a further notional extension, as they are polymorphic: for instance, (CON′ a) is given the notional type (*level* i) for any i :: *nat*.

Thus, we also notionally translate *subtype polymorphism* (see [62, ch. 15]), with (level i) being a subtype of (level j) when $\mathrm{i} \leqslant \mathrm{j}$. That is the content of Lemma 3.3; however, it is not usable directly as a conditional rewrite rule because it would loop. Fortunately, its non-looping corollary

$$\mathsf{level}\ \mathrm{i}\ \mathrm{s} \Longrightarrow \mathsf{level}\ (\mathrm{i} + 1)\ \mathrm{s}$$

is sufficient for those cases where we will need explicit use of subsumption, specifically for proofs by induction on the level i.

Lemmas providing additional simplifier rules for level will be proved as further constructs of type *dB* are introduced. We have already seen one such construct, the function dB :: (*expr* ⇒ *dB*); the corresponding rule follows from Definition 3.4 and Lemma 3.3:

Lemma 3.8

level i (dB x)

Extending the translation to functional types with negative occurrences of (*level* i) might complicate matters; fortunately, we will only need positive occurrences, for which the extension to functional types is straightforward.

All versions of Hybrid follow a general pattern of making definitions and proving lemmas first for arbitrary levels, and then deriving the desired results for proper terms as corollaries. In the present version, arbitrary levels are handled by recursion and induction over de Bruijn syntax, using the type *dB* and the predicate level as described above, while the results for proper terms are stated at type *expr*.

3.2.4 Definition of "abstr" and "LAM"

We now turn to the task of defining abstr and LAM. The main ideas are from [2], but the details of the definitions and proofs are original. There are some improvements over the original version of Hybrid, which will be described in this section and Section 3.2.6.

Since we will be defining abstr and LAM in terms of de Bruijn syntax, the definition of syntactic functions from Section 3.1 is not directly usable here: we need an analogous definition using de Bruijn syntax in place of LAM.

For recursion, we must work with *dB*-valued functions (arbitrary levels) rather than *expr*-valued functions. However, the argument type need not also be *dB*, and

in fact it will be more convenient to work with functions of type (*expr* ⇒ *dB*). This avoids negative occurrences of the type *dB*, and therefore of the notional type (*level* i) as described in Section 3.2.3.

Thus we define the *syntactic dB-terms*, as a subset of Isabelle/HOL terms of type *dB*, using variables of type *expr* converted via dB:

$$s ::= \mathsf{dB}\ x \mid \mathsf{CON'}\ a \mid \mathsf{VAR'}\ n \mid s_1\ \$\$'\ s_2 \mid \mathsf{ERR'} \mid \mathsf{BND'}\ i \mid \mathsf{ABS'}\ s$$

where s (with possible subscripts) stands for a syntactic *dB*-term, x for a variable of type *expr*, a for a constant of type *con*, and n and i for natural-number constants. We define the *syntactic dB-functions* as the functions of type (*expr* ⇒ *dB*) of the form (λx. s), where s is a syntactic *dB*-term with (at most) one free variable x. Such functions mix de Bruijn indices (BND′) with higher-order abstract syntax (using the Isabelle/HOL bound variable x to represent an object-language variable).

We define a predicate Abstr to recognize the syntactic *dB*-functions, together with an auxiliary predicate ordinary needed in the definition of Abstr:

Definition 3.9

ordinary :: (*b* ⇒ *a dB*) ⇒ *bool*

ordinary X ≡ (∃ a. X = (λ x. CON′ a)) ∨ (∃ n. X = (λ x. VAR′ n)) ∨
(∃ S T. X = (λ x. S x $$′ T x)) ∨ (X = (λ x. ERR′)) ∨
(∃ j. X = (λ x. BND′ j)) ∨ (∃ S. X = (λ x. ABS′ (S x)))

Definition 3.10

function Abstr :: (*a expr* ⇒ *a dB*) ⇒ *bool*

Abstr (λ x. CON′ a) = True
Abstr (λ x. VAR′ n) = True
Abstr (λ x. S x $$′ T x) = (Abstr S ∧ Abstr T)
Abstr (λ x. ERR′) = True
Abstr (λ x. BND′ i) = True
Abstr (λ x. ABS′ (S x)) = Abstr S
¬ ordinary S ⟹ Abstr S = (S = dB)

Syntactically, the defining equations for Abstr have the form of recursion on the *body* of a λ-abstraction. Mathematically, they define (Abstr S) by recursion on the *common structure* of all the values of the function S, i.e., on the common structure (if any) of (S x) for all x :: *expr*. The predicate ordinary is defined to recognize those functions that match one of the first six equations, so that the condition ($\neg$ ordinary S) on the last equation may be read as "otherwise"; that equation corresponds to the variable case for syntactic *dB*-terms as defined above.

The **function** command demands proofs of pattern completeness and compatibility: that is, every function S :: (*expr* $\Rightarrow$ *dB*) must match at least one of the defining equations, and if two equations match the same function S, then they must give the same value for (Abstr S). Here, pattern completeness follows from the definition of ordinary, while compatibility is trivial as the cases are disjoint.

It is followed by a **termination** command (not shown), which demands a termination order for the recursion and a proof that it is well-founded. Here we use (size (S arbitrary)) as a termination measure: the size of one value of the function S serves as an upper bound for the size of the common structure of all values of S.[7]

Once these proof obligations are satisfied, Isabelle/HOL defines the constant Abstr and proves its defining equations as formal theorems.[8] It also provides structural induction and case-distinction rules for the type (*expr* $\Rightarrow$ *dB*), called Abstr.induct and Abstr.cases respectively, based on the pattern of recursion used in the definition.

We may now define the predicate abstr in terms of Abstr by using post-composition with dB to convert its function argument from the type (*expr* $\Rightarrow$ *expr*) to (*expr* $\Rightarrow$ *dB*).

[7] Isabelle/HOL's polymorphic constant arbitrary gives a fixed but arbitrary element of any type.

[8] In fact the constant is defined by the **function** command, even without a termination order; a recursion that does not always terminate gives a function that is undefined on some arguments.

Definition 3.11

abstr :: $(a\ expr \Rightarrow a\ expr) \Rightarrow bool$
abstr S ≡ Abstr (dB ∘ S)

Note that unlike the situation in [2], the definition of Abstr does not need to impose a constraint on the argument of BND′, because in the case of (abstr S) dangling indices are excluded by the type of the function S :: $(expr \Rightarrow expr)$.

Lemma 3.12

Abstr_const: Abstr (λ x. s)

The lemma Abstr_const shows that any constant function of type $(expr \Rightarrow dB)$ satisfies Abstr. It is used to prove a similar property for abstr, and will later be used directly as well. It is proved by induction on s using Definition 3.10 (Abstr).

Lemma 3.13

abstr_id: abstr (λ x. x)
abstr_const: abstr (λ x. s)
abstr_APP: abstr (λ x. S x $$ T x) = (abstr S ∧ abstr T)

The lemma abstr_const is a corollary of Abstr_const, while the other two lemmas are proved directly, using Definitions 3.11 (abstr) and 3.10 (Abstr).

These lemmas allow abstr conditions for syntactic functions to be proved compositionally without unfolding the definition, except when the body of the function contains a LAM subterm that involves the function argument (so that it is not just a constant). In that case, previous versions of Hybrid required unfolding the definitions of abstr and LAM to convert HOAS to de Bruijn syntax. The present work improves on that situation by providing a compositional rule also for the LAM case (Lemma 3.38 in Section 3.2.9).

The lemma abstr_const will be important for Hybrid terms with nested LAM operators, to show that the argument of an inner LAM satisfies abstr when its body

contains a bound variable from an outer LAM; such a bound variable is a placeholder for an arbitrary term of type *expr*, which is exactly the role of s in abstr_const.

We now define the function LAM, using the same form of recursion that was used in the definition of abstr.

Definition 3.14

LAM :: $(a\ expr \Rightarrow a\ expr) \Rightarrow a\ expr$
  LAM S $\equiv$ expr (Lambda (dB $\circ$ S))
Lambda :: $(a\ expr \Rightarrow a$ dB$) \Rightarrow a\ dB$
  Lambda S $\equiv$ if (Abstr S) then (ABS′ (Lbind 0 S)) else ERR′

The function LAM, like abstr, first composes dB with the given function. It then applies the auxiliary function Lambda and converts the resulting term from type *dB* to type *expr*.

The function Lambda first checks if its argument satisfies Abstr, and produces ERR′ if not. (This is equivalent to checking if the argument of LAM satisfies abstr.) The original version of Hybrid [2] did not do this check (and did not have the constant ERR′), making it impossible to determine from (LAM S) whether S is a syntactic function or not. We include these features to support the stronger injectivity property for LAM proved in Section 3.2.6.

If its argument does satisfy Abstr, then Lambda applies another auxiliary function Lbind, defined by recursion, to convert HOAS to de Bruijn syntax; i.e., to convert the variable represented by the function argument into a dangling de Bruijn index. It then applies a new ABS′ node to bind the variable and obtain a proper de Bruijn term.

Definition 3.15

function Lbind :: $\left[\, bnd, (a\ expr \Rightarrow a\ dB) \,\right] \Rightarrow a\ dB$
  Lbind i (λ x. CON′ a) = CON′ a
  Lbind i (λ x. VAR′ n) = VAR′ n
  Lbind i (λ x. S x $$′ T x) = Lbind i S $$′ Lbind i T
  Lbind i (λ x. ERR′) = ERR′

Lbind i (λ x. BND′ j) = BND′ j
Lbind i (λ x. ABS′ (S x)) = ABS′ (Lbind (i + 1) S)
$\neg$ ordinary S $\Longrightarrow$ Lbind i S = BND′ i

The auxiliary function Lbind extracts the common structure of the values of its function argument, replacing indecomposable uses of the bound variable (i.e., functions that do not match any of the first six equations) with (BND′ i). This is a dangling de Bruijn index, and i is incremented each time the recursion passes an ABS′ node so that all such instances of BND′ will refer to the ABS′ node added by Lambda. The Abstr condition checked in the definition of Lambda ensures that the last equation will be applied only when S = (λ x. dB x).

Lemma 3.16

Lbind_const: Lbind i (λ x. s) = s

The lemma Lbind_const shows that applying (Lbind i) to a constant function of type ($expr \Rightarrow dB$) gives the constant value of that function. It is proved by induction on s. This lemma will be important for Hybrid terms with nested LAM operators, to allow the argument of an outer LAM to satisfy abstr when its bound variable occurs in the scope of an inner LAM.

Since Lbind is a new construct of type dB, we need a simplifier rule for level applied to it. We first define an abbreviation Level for pointwise application of level to a function:

Definition 3.17

abbreviation Level :: $\left[bnd, (b \Rightarrow a\ dB) \right] \Rightarrow bool$
Level i S $\equiv \forall$ x. level i (S x)

This abbreviation is then used in the premise of a conditional rewrite rule:

Lemma 3.18

Level_Lbind: Level i S $\Longrightarrow$ level (i + 1) (Lbind i S)

This lemma was proved from the definition of Lbind by induction (Abstr.induct). Stronger results are possible, but this one will be sufficient.

Lemma 3.19

dB_LAM: dB (LAM S) = if (abstr S) then (ABS′ (Lbind 0 (dB ∘ S))) else ERR′
abstr_dB_LAM: abstr S ⟹ dB (LAM S) = ABS′ (Lbind 0 (dB ∘ S))

The lemma dB_LAM combines unfolding of Definition 3.14 (LAM and Lambda) with cancellation of the functions dB and expr, using the fact that both ERR′ and (ABS′ (Lbind 0 (dB ∘ S))) are proper. (Dangling indices are excluded from S :: (*expr* ⇒ *expr*) by its type, and the one introduced by Lbind is bound by the enclosing ABS′.)

The lemma abstr_dB_LAM is a weaker version intended as a conditional rewrite rule for Isabelle's simplifier, to do the unfolding only if the abstr condition simplifies to True.

With the definitions above, Hybrid terms using LAM (i.e., closed syntactic terms) are provably equal to the corresponding de Bruijn syntax representations, converted to the type *expr* using the function expr. (This is much the same situation as in [2], except for the type conversion which was not necessary there.) Thus, starting from two *distinct* representations for free variables, we have established two *ambiguous* representations for bound variables, in the sense that any given element of *expr* may be viewed as having either form. (An example is given in Section 3.2.5.) In the remainder of Section 3.2, we will state results using the HOAS representation (LAM) but use the de Bruijn syntax representation (ABS′/BND′) in proofs by induction, aiming to characterize the former representation so that it stands on its own.

The fact that abstr as defined above agrees with the definition of syntactic functions given in Section 3.1 will follow from Theorem 3.56 (Adequacy).

All versions of Hybrid have used essentially the same form of recursion to define abstr and LAM, and the corresponding form of induction to prove their properties.

However, the means of formalizing it have varied greatly:

- The original version [2] used inductively-defined predicates and induction on those predicates. However, for LAM it was necessary to convert a predicate lbnd to a function lbind (corresponding to our Lbind) using a description operator. This required several technical lemmas, including an induction principle called abstraction_induct (corresponding to our Abstr.induct), proved by size induction (corresponding to our termination measure).

- The following version [18, 51] defined a polymorphic datatype *dB_fn* with constructors corresponding to the cases of Definition 3.10, and used a generalized version of abstraction_induct (together with a description operator) to establish inverse bijections between a subset of *dB_fn* and the *dB*-valued functions. It then used primitive recursion and induction on *dB_fn* as supplied by Isabelle/HOL's datatype package. This approach simplified the definition of LAM and some of the proofs, but the use of an auxiliary datatype was itself a significant complication.

- The present version uses the function package introduced in Isabelle/HOL 2007, which directly supports the desired form of recursion and thus avoids many of the complications of the previous approaches, even proving a corresponding induction principle (Abstr.induct) automatically.

A predicate called ordinary has also been present in all versions of Hybrid, though it originally included the variable case as well.[9] Removing this case allowed ordinary to be generalized to *dB*-valued functions on any type; this will allow us to reuse it for binary functions in Section 3.2.9 and for n-ary functions in Section 3.3.

[9] We retain the predicate name ordinary for consistency with [2], although "head-constant" would be a more descriptive term.

3.2.5 Example: "abstr" and "LAM"

To illustrate the definitions of abstr and LAM, and some of the complications that arise from nesting of LAM, we consider a simple example of a Hybrid term:

LAM x. LAM y. x $$ y

The argument of the inner LAM, (λ y. x $$ y), can be proved to satisfy abstr using Lemma 3.13, for any x :: *expr*:

abstr (λ y. x)	(by abstr_const)
abstr (λ y. y)	(by abstr_id)
abstr (λ y. (x $$ y))	(by abstr_APP)

We may then unfold Definition 3.14 for the inner LAM, immediately simplifying the conditional construct:

$$\begin{array}{ll} & \mathsf{LAM}\ y.\ x\ \$\$\ y \\ \equiv & \mathsf{expr}\ (\mathsf{ABS}'\ (\mathsf{Lbind}\ 0\ (\mathsf{dB} \circ (\lambda\ y.\ x\ \$\$\ y)))) \end{array}$$

The conversion from HOAS in the argument of LAM to de Bruijn syntax goes in three stages. First the function of type (*expr* $\Rightarrow$ *expr*) is converted to (*expr* $\Rightarrow$ *dB*) by composing dB with it:

$$\begin{array}{ll} & \mathsf{dB} \circ (\lambda\ y.\ x\ \$\$\ y) \\ \equiv & \lambda\ y.\ \mathsf{dB}\ (x\ \$\$\ y) \\ \equiv & \lambda\ y.\ \mathsf{dB}\ x\ \$\$'\ \mathsf{dB}\ y \end{array}$$

(The function dB remains where it is applied to a variable.) Next, the function of type (*expr* $\Rightarrow$ *dB*) is converted to a term of type *dB* using Lbind, which replaces the HOAS variable in the form (dB y) with a dangling de Bruijn index:

$$\begin{array}{ll} & \mathsf{Lbind}\ 0\ (\lambda\ y.\ \mathsf{dB}\ x\ \$\$'\ \mathsf{dB}\ y) \\ \equiv & \mathsf{Lbind}\ 0\ (\lambda\ y.\ \mathsf{dB}\ x)\ \$\$'\ \mathsf{Lbind}\ 0\ (\lambda\ y.\ \mathsf{dB}\ y) \\ \equiv & \mathsf{dB}\ x\ \$\$'\ \mathsf{BND}'\ 0 \end{array}$$

(The first Lbind is simplified by Lemma 3.16 (Lbind_const), while the second one η-contracts to (Lbind 0 dB) which is (BND′ 0) by Definition 3.15.) Finally, this term

is wrapped in ABS′ to bind the dangling index, and converted to the type *expr* using expr; in summary, we have shown

(LAM y. x $$ y) = expr (ABS′ (dB x $$′ BND′ 0)).

Or using Lemma 3.19 (abstr_dB_LAM),

dB (LAM y. x $$ y) = ABS′ (dB x $$′ BND′ 0).

The argument of the outer LAM can now be shown to be satisfy abstr by unfolding Definition 3.11 and using the defining equations for Abstr (Definition 3.10). This is no longer the recommended way; we will revisit this example in Section 3.2.9, after establishing a compositional rule (Lemma 3.38) for simplifying the LAM case of abstr.

$$
\begin{aligned}
& \mathsf{abstr}\ (\lambda\ \mathrm{x}.\ \mathsf{LAM}\ \mathrm{y}.\ \mathrm{x}\ \$\$\ \mathrm{y}) \\
\equiv\ & \mathsf{Abstr}\ (\mathsf{dB} \circ (\lambda\ \mathrm{x}.\ \mathsf{LAM}\ \mathrm{y}.\ \mathrm{x}\ \$\$\ \mathrm{y})) \\
\equiv\ & \mathsf{Abstr}\ (\lambda\ \mathrm{x}.\ \mathsf{ABS}'\ (\mathsf{dB}\ \mathrm{x}\ \$\$'\ \mathsf{BND}'\ 0)) \\
\equiv\ & \mathsf{Abstr}\ (\lambda\ \mathrm{x}.\ \mathsf{dB}\ \mathrm{x}\ \$\$'\ \mathsf{BND}'\ 0) \\
\equiv\ & \mathsf{Abstr}\ (\lambda\ \mathrm{x}.\ \mathsf{dB}\ \mathrm{x}) \wedge \mathsf{Abstr}\ (\lambda\ \mathrm{x}.\ \mathsf{BND}'\ 0) \\
\equiv\ & \mathsf{True} \wedge \mathsf{True} \quad \equiv \quad \mathsf{True}
\end{aligned}
$$

(The first Abstr η-contracts to (Abstr dB) which is true by Definition 3.10, while Lemma 3.12 (Abstr_const) proves the second one.)

For the final part of the example, we convert the original term with nested LAM operators fully to de Bruijn syntax, by unfolding Definition 3.14 for the outer LAM:

$$
\begin{aligned}
& \mathsf{LAM}\ \mathrm{x}.\ \mathsf{LAM}\ \mathrm{y}.\ \mathrm{x}\ \$\$\ \mathrm{y} \\
\equiv\ & \mathsf{expr}\ (\mathsf{ABS}'\ (\mathsf{Lbind}\ 0\ (\mathsf{dB} \circ (\lambda\ \mathrm{x}.\ \mathsf{LAM}\ \mathrm{y}.\ \mathrm{x}\ \$\$\ \mathrm{y})))) \\
\equiv\ & \mathsf{expr}\ (\mathsf{ABS}'\ (\mathsf{Lbind}\ 0\ (\lambda\ \mathrm{x}.\ \mathsf{ABS}'\ (\mathsf{dB}\ \mathrm{x}\ \$\$'\ \mathsf{BND}'\ 0)))) \\
\equiv\ & \mathsf{expr}\ (\mathsf{ABS}'\ (\mathsf{ABS}'\ (\mathsf{Lbind}\ 1\ (\lambda\ \mathrm{x}.\ \mathsf{dB}\ \mathrm{x}\ \$\$'\ \mathsf{BND}'\ 0)))) \\
\equiv\ & \mathsf{expr}\ (\mathsf{ABS}'\ (\mathsf{ABS}'\ (\mathsf{Lbind}\ 1\ (\lambda\ \mathrm{x}.\ \mathsf{dB}\ \mathrm{x})\ \$\$'\ \mathsf{Lbind}\ 1\ (\lambda\ \mathrm{x}.\ \mathsf{BND}'\ 0)))) \\
\equiv\ & \mathsf{expr}\ (\mathsf{ABS}'\ (\mathsf{ABS}'\ (\mathsf{BND}'\ 1\ \$\$'\ \mathsf{BND}'\ 0)))
\end{aligned}
$$

(The first Lbind η-contracts to (Lbind 1 dB) = (BND′ 1), while the second one is simplified by Lbind_const.)

The type conversions dB and expr complicate this example, as compared with the original version of Hybrid [2] which did not have the **typedef** of Definition 3.4. However, in practice these details are handled automatically by Isabelle's simplifier; and in object-language work using Hybrid, there is no need to unfold the definitions or work with de Bruijn syntax at all. The benefits of eliminating explicit proper conditions and strengthening the injectivity property for LAM (Theorem 3.21) make the **typedef** worthwhile.

3.2.6 Injectivity of "LAM"

As stated in Section 3.1, Hybrid proves injectivity of LAM restricted to functions of type ($expr \Rightarrow expr$) satisfying abstr. Improving on [2], this property is strengthened by requiring only one abstr premise, using the fact that LAM maps functions not satisfying abstr to a recognizable placeholder term ERR.

We begin with an injectivity result for arbitrary de Bruijn levels.

Lemma 3.20

Abstr_Lbind_inject:

$[\![\, \mathsf{Abstr}\ \mathrm{S};\ \mathsf{Abstr}\ \mathrm{S}';\ \mathsf{Level}\ \mathrm{i}\ \mathrm{S};\ \mathsf{Level}\ \mathrm{i}\ \mathrm{S}' \,]\!] \Longrightarrow (\mathsf{Lbind}\ \mathrm{i}\ \mathrm{S} = \mathsf{Lbind}\ \mathrm{i}\ \mathrm{S}') = (\mathrm{S} = \mathrm{S}')$

This lemma is proved by a straightforward induction on S :: ($expr \Rightarrow dB$) using Abstr.induct (from Definition 3.10).

Theorem 3.21 (Injectivity of LAM)

$[\![\, \mathsf{LAM}\ \mathrm{S} = \mathsf{LAM}\ \mathrm{T};\ \mathsf{abstr}\ \mathrm{S} \vee \mathsf{abstr}\ \mathrm{T} \,]\!] \Longrightarrow \mathrm{S} = \mathrm{T}$

Proof. If one of S and T satisfies abstr and the other does not, then by Lemma 3.19 (dB_LAM), one of the terms (dB (LAM S)) and (dB (LAM T)) is of the form (ABS′ t) for some t :: dB, while the other is ERR′. But these terms cannot be equal, which contradicts the premise LAM S = LAM T. Thus the original assumption must be false, and we must have both (abstr S) and (abstr T).

We apply dB to both sides of the equality LAM S = LAM T and simplify using abstr_dB_LAM (Lemma 3.19) to obtain

ABS′ (Lbind 0 (dB ∘ S)) = ABS′ (Lbind 0 (dB ∘ T)).

ABS′ is a datatype constructor and thus injective, so we may cancel it:

Lbind 0 (dB ∘ S) = Lbind 0 (dB ∘ T).

We have (Abstr (dB ∘ S)) and (Abstr (dB ∘ T)) by unfolding Definition 3.11 (abstr), and we have (Level 0 (dB ∘ S)) and (Level 0 (dB ∘ T)) since terms converted from type *expr* are proper by Lemma 3.8. Thus we may apply the preceding lemma (Abstr_Lbind_inject) to deduce dB ∘ S = dB ∘ T. Since dB is injective, it can be canceled to obtain S = T, as was to be proven. □

Note that (Lbind 0) is only injective on functions from *expr* to *dB* whose values are proper terms, i.e., those that factor through dB, because any pre-existing dangling indices at level 1 would be indistinguishable from those resulting from conversion of the HOAS variable. For example,

Lbind 0 (λ x. dB x) = BND′ 0 = Lbind 0 (λ x. BND′ 0).

Thus, without the **typedef** limiting *expr* to proper terms, we would not be able to avoid conditions on both S and T; at best, we could replace one abstr condition with something like (∀ x. proper x ⟶ proper (T x)).

Attempts were made to work around this problem, prior to the introduction of a type of proper terms, by excluding dangling indices in the definition of abstr or LAM. However, it was found that any such change invalidated either Abstr_const (Lemma 3.12) or Lbind_const (Lemma 3.16), causing the argument of at least one of the LAM operators in (LAM x. LAM y. x $$ y) to fail to satisfy abstr. In effect, abstr and LAM must concern themselves *only* with the behaviour of the bound variable, and not with the structure of any constant part of the function body, as the latter could be the bound variable of an enclosing LAM.

The advantage of an injectivity property that can work with a condition on only one of S and T is that it simplifies the elimination rules for inductively-defined predicates on Hybrid terms, such as those found in Section 5.3 (representing an evaluation judgment for Mini-ML with references). As a result, abstr conditions are more often available where they are needed, without having to add them as premises.

We now have everything needed to state the distinctness and injectivity properties for *expr* promised in Section 3.1.

Lemma 3.22 (Distinctness of Hybrid operators)

expr_distinct:

VAR n ≠ CON a s $$ t ≠ CON a s $$ t ≠ VAR n
LAM S ≠ CON a LAM S ≠ VAR n LAM S ≠ s $$ t
ERR ≠ CON a ERR ≠ VAR n ERR ≠ s $$ t
abstr S ⟹ ERR ≠ LAM S

Lemma 3.23 (Injectivity of Hybrid operators)

expr_inject:

(CON a_1 = CON a_2) = (a_1 = a_2)
(VAR n_1 = VAR n_2) = (n_1 = n_2)
(s $$ t = u $$ v) = (s = u ∧ t = v)
abstr S ⟹ (LAM S = LAM T) = (S = T)
abstr T ⟹ (LAM S = LAM T) = (S = T)

All of these properties are declared as simplification rules, so that Isabelle's **auto** and **simp** proof methods will use them automatically. Thus, they rarely appear explicitly in proofs of properties of object languages represented using Hybrid.

3.2.7 Conversion from de Bruijn indices to HOAS

We have seen in Sections 3.2.4 and 3.2.5 how the definitions of LAM and abstr convert a HOAS representation of variable binding to one based on de Bruijn indices. The

reverse process is also useful, e.g., to prove exhaustiveness of Hybrid's operators for the type *expr*.

The original version of Hybrid [2] did this using a function called lconv, with simplifier rules that fully convert a de Bruijn term to HOAS in one pass. We use a simpler variant of the same idea, converting just one ABS′ into LAM since that is sufficient for our purposes.

Definition 3.24

fun Linv :: $\left[bnd,\ a\ dB \right] \Rightarrow (a\ expr \Rightarrow a\ dB)$
  Linv i (CON′ a) = (λ x. CON′ a)
  Linv i (VAR′ n) = (λ x. VAR′ n)
  Linv i (s $$′ t) = (λ x. (Linv i s) x $$′ (Linv i t) x)
  Linv i ERR′ = (λ x. ERR′)
  Linv i (BND′ j) = (if j = i then dB else (λ x. BND′ j))
  Linv i (ABS′ s) = (λ x. ABS′ ((Linv (i + 1) s) x))

The function Linv is the reverse of Lbind, converting a specified dangling de Bruijn index to a HOAS representation of a variable, specifically the variable case (dB x) from the definition of syntactic *dB*-functions in Section 3.2.4. Several basic properties are proved as lemmas:

Lemma 3.25

Linv_inverse: Lbind i (Linv i s) = s
Abstr_Linv: Abstr (Linv i s)
Level_Linv: level (i + 1) s ⟹ Level i (Linv i s)
Linv_const: level i s ⟹ Linv i s = (λ x. s)

The lemma Linv_inverse shows that (Linv i) is pre-inverse to (Lbind i), for any i :: *bnd*. It is not a full inverse function, as (Lbind i) leaves no way to distinguish pre-existing dangling indices from newly introduced ones, as mentioned previously. (With a Level condition to exclude dangling indices, it would be post-inverse too; but that property will not be needed.)

The lemma Abstr_Linv shows that (Linv i) produces syntactic *dB*-functions. The lemma Level_Linv shows that (Linv i) applied to a term of level i + 1 produces a result of level i, by replacing the dangling indices that were responsible for the higher level. The lemma Linv_const shows that (Linv i) applied to a term of level i produces a constant function whose constant value is that term.

All four lemmas were proved by straightforward inductions on s.

Lemma 3.26

Abstr_Linv_Abstr: Abstr S $\Longrightarrow$ Abstr (λ x. (Linv i (S x)) y)

The lemma Abstr_Linv_Abstr shows that if (S x) is a syntactic *dB*-function of x, then so is ((Linv i (S x)) y) for any fixed i :: *nat* and y :: *expr*. It is proved by induction on S (using Abstr.induct from Definition 3.10), with the help of Linv_const from Lemma 3.25.

Lemma 3.27

expr_ABS′_LAM: level 1 s $\Longrightarrow$ expr (ABS′ s) = LAM (expr $\circ$ (Linv 0 s))

The lemma expr_ABS′_LAM shows that expr applied to an ABS′ term is equal to LAM applied to a certain function constructed with the help of Linv.

In addition to using conversion from de Bruijn syntax to HOAS to prove exhaustiveness and induction, the original version of Hybrid [2] specified methods for performing this conversion and its inverse on concrete terms. Conversion from HOAS to de Bruijn syntax simply required unfolding the definition of LAM and using simplifier rules. Conversion in the other direction, however, required inserting a trivial instance of the auxiliary function lconv of type $dB \Rightarrow dB$, and this was hard to control.

The present version of Hybrid aims to make it unnecessary to perform such conversions by keeping object-language work entirely at the HOAS level. But for demonstration purposes, it supports automatic conversion between HOAS and de Bruijn syntax controlled by the type-conversion functions dB and expr. Simplifier rules are

provided so that Isabelle's **simp** and **auto** proof methods will rewrite

dB (LAM x. LAM y. x $$ y) to ABS′ (ABS′ (BND′ 1 $$′ BND′ 0)),

and conversely

expr (ABS′ (ABS′ (BND′ 1 $$′ BND′ 0))) to LAM x. LAM y. x $$ y.

Having separate types *expr* for HOAS and *dB* for de Bruijn syntax makes these conversions easier to control. (The latter conversion does, however, require more rewrite steps than the one-pass conversion provided by the original version of Hybrid [2].)

The main rewrite rules are Lemmas 3.6 and 3.7 for the non-HOAS operators, together with Lemmas 3.19 (abstr_dB_LAM) and 3.27 (expr_ABS′_LAM) for conversion between LAM and ABS′. Supporting rules are provided for proving abstr conditions (Lemma 3.13 and the upcoming Lemma 3.38), proving level conditions, and simplifying Lbind and Linv, including Lemmas 3.16 (Lbind_const) and 3.25 (Linv_const) in addition to the defining equations.

The rule expr_ABS′_LAM is added to the default simpset only at the end of the theory file, since expr applied to a partially specified ABS′ term occurs in several places in Hybrid's lemmas, and attempting to convert such a term to HOAS would be a step in the wrong direction. Conversely, injectivity of expr (on proper terms) is removed from the default simpset at the end of the theory file, as it is useful within Hybrid but it would interfere with the use of expr to convert de Bruijn syntax to HOAS. These adjustments should be reversed when developing extensions of Hybrid as separate theory files, as is done in Section 3.3.

3.2.8 Exhaustiveness and induction

The original version of Hybrid [2] proved a case analysis rule called properE for terms satisfying a proper condition, and an induction rule using VAR to represent open terms, which was proved in turn using size induction. We adapt these features to our type of proper terms *expr*.

Lemma 3.28

expr_nchotomy: $(\exists\ a.\ y = \mathsf{CON}\ a) \vee (\exists\ n.\ y = \mathsf{VAR}\ n) \vee (\exists\ s\ t.\ y = s\ \$\$\ t)$
$\vee\ (\exists\ S.\ \mathsf{abstr}\ S \wedge y = \mathsf{LAM}\ S) \vee y = \mathsf{ERR}$

The lemma expr_nchotomy shows that a term y :: *expr* must have one of the five forms (CON a), (VAR n), (s $$ t), (LAM S) where S satisfies abstr, or ERR.

Proof. By Definition 3.4 (*expr*), we must have $y = \mathsf{expr}\ y'$ for some y' :: *dB* satisfying (level 0 y'). Then y' must be formed using one of the constructors of *dB*. It cannot be (BND′ j), as (level i (BND j)) is true only when i > j, and we have (level 0 y'). If it is (ABS′ s'), we rewrite $y = \mathsf{expr}\ (\mathsf{ABS}'\ s')$ using Lemma 3.27 (expr_ABS′_LAM) to put y in the form (LAM S). Otherwise, one of the equations from Lemma 3.7 will apply to put y in one of the four other forms listed. □

This fact is also stated as an elimination rule called expr_exhaust suitable for use with Isabelle's **cases** proof method, similar to the original version's properE. The transformation from disjunction to elimination rule is straightforward, and we omit the statement. (The suffixes "nchotomy" and "exhaust" follow the pattern used by Isabelle/HOL's **datatype** package.)

In the present version of Hybrid, we add a similar case analysis rule for syntactic functions. Only the elimination-rule form is stated at this point, since we will later strengthen the disjunction form to a biconditional (Lemma 3.39, expand_abstr).

Lemma 3.29

abstr_cases:
$[\![\ \mathsf{abstr}\ Y;\quad [\![Y = (\lambda\ x.\ x)]\!] \Longrightarrow P;$
$\bigwedge a.\ [\![Y = (\lambda\ x.\ \mathsf{CON}\ a)]\!] \Longrightarrow P;$
$\bigwedge n.\ [\![Y = (\lambda\ x.\ \mathsf{VAR}\ n)]\!] \Longrightarrow P;$
$\bigwedge S\ T.\ [\![Y = (\lambda\ x.\ S\ x\ \$\$\ T\ x);\ \mathsf{abstr}\ S;\ \mathsf{abstr}\ T]\!] \Longrightarrow P;$
$\bigwedge W.\ [\![Y = (\lambda\ x.\ \mathsf{LAM}\ y.\ W\ x\ y);$
$(\forall\ x.\ \mathsf{abstr}\ (\lambda\ y.\ W\ x\ y));\ (\forall\ y.\ \mathsf{abstr}\ (\lambda\ x.\ W\ x\ y))]\!] \Longrightarrow P;$
$[\![Y = (\lambda\ x.\ \mathsf{ERR})]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$

This lemma is also proved by using Lemma 3.27 (expr_ABS′_LAM) to convert ABS′ to LAM, this time after unfolding Definitions 3.11 (abstr) and 3.10 (Abstr). Establishing the abstr conditions for the ABS′/LAM case also requires Lemma 3.26 (Abstr_Linv_Abstr), which was not needed in the proof of Lemma 3.28.

In [2], a structural induction rule called proper_VAR_induct was proved by size induction. We do the same, but since our *expr* is not a datatype, it is first necessary to define a size function for it and prove its basic properties.

Definition 3.30

size (y :: *a expr*) ≡ size (dB y)

The constant size is actually a type-class constant in Isabelle/HOL; this definition instantiates the type class SIZE, which consists of the constant size :: $a \Rightarrow nat$ with no axioms, for *expr*. The size of a Hybrid term s :: *expr* is defined to be the size of (dB s) :: *dB*, which is a term of a datatype, for which size is automatically defined by Isabelle/HOL.

Lemma 3.31

size_ERR: size ERR = 0
size_CON: size (CON a) = 0
size_VAR: size (VAR n) = 0
size_APP: size (s $$ t) = size s + size t + 1

These properties follow immediately from Definition 3.30, using Lemma 3.6 (rewrite rules for dB) and the properties of size :: ($dB \Rightarrow nat$) provided by the **datatype** package. They are declared as rewrite rules for Isabelle's simplifier, to support size induction.

Lemma 3.32

size_LAM: ⟦ size t = 0; abstr S ⟧ ⟹ size (LAM S) = size (S t) + 1
size_LAM_lt: ⟦ size t = 0; abstr S ⟧ ⟹ size (S t) < size (LAM S)

The lemma size_LAM is proved by Abstr.induct (from Definition 3.10) on (dB ∘ S), after unfolding Definition 3.11 (abstr) and Definition 3.30 and allowing the simplifier rules for dB to unfold the definition of LAM. It shows that the size of (S t) is one less than the size of (LAM S), if S satisfies abstr and t has size 0.

Because of the unknown t on the right-hand side of the equality, size_LAM is not usable as a simplifier rule. Thus we add its corollary size_LAM_lt to the simpset instead.

Lemma 3.33

expr_VAR_induct:
⟦ ⋀ a. P (CON a); ⋀ n. P (VAR n); ⋀ s t. ⟦ P s; P t ⟧ ⟹ P (s $$ t);
⋀ S. ⟦ abstr S; ∀ n. P (S (VAR n)) ⟧ ⟹ P (LAM S); P ERR ⟧ ⟹ P y

The lemma expr_VAR_induct is a structural induction rule for *expr*, which represents open terms using VAR and corresponds to proper_VAR_induct from [2] (without the proper premise since it is implicit in the type *expr*). It is proved by size induction on y :: *expr*, using case distinction on y (Lemma 3.28, expr_nchotomy) and the simplifier rules we have defined for size :: (*expr* ⇒ *nat*).

To actually use this induction rule effectively would require auxiliary definitions and lemmas for reasoning about free variables. The original version of Hybrid provided these things, but such reasoning is tedious; we did not adapt these features for the present version of Hybrid, instead seeking to develop a better alternative, as described in Section 3.3.

3.2.9 Characterizing "abstr"

In Section 3.2.4, an incomplete set of simplification rules for abstr was provided as Lemma 3.13. The missing case is (abstr (λ x. LAM y. W x y)).

Both previous versions of Hybrid [2, 18, 51] relied on conversion from HOAS to de Bruijn syntax to handle this case. That is sufficient for proving that particular

syntactic functions satisfy abstr, as seen in Section 3.2.5; but it is less useful for partially-specified functions as found in inductive proofs.

We could obtain a compositional introduction rule for this case by defining a predicate biAbstr :: $\big([expr,\ expr] \Rightarrow expr\big) \Rightarrow bool$ generalizing abstr, and proving

biAbstr W $\Longrightarrow$ abstr (λ x. LAM y. W x y).

This was done by Momigliano et al. [50]; their formal theory BiAbstr is available online [18].

However, the LAM case arises again for biAbstr, and for any higher-arity generalization. There are several ways to address this:

- Use Isabelle/HOL's axiomatic type classes to define a polymorphic predicate generalizing abstr to curried functions of arbitrary arity. This looks like a promising approach, but it remains as future work.
- Find a single type that can represent functions of arbitrary arity, and generalize Hybrid's constructs to that type. Some experimental work in that direction is the subject of Section 3.3. (Such a type is also useful as a representation of open terms for induction.)
- Prove a result that reduces biAbstr to abstr. This seems to be the most direct solution, and it is the approach we take in the present work.

In this section, we will represent functions of two arguments using pairs, rather than in the usual curried form, so that we may reuse Definition 3.9 (ordinary) and some technical lemmas (left unstated as they are mathematically trivial), all of which refer to the polymorphic type $(b \Rightarrow dB)$.

Definition 3.34

abstr_2 :: $(a\ expr \times a\ expr \Rightarrow a\ expr) \Rightarrow bool$
  abstr_2 S $\equiv$ Abstr_2 (dB $\circ$ S)

The predicate abstr_2 generalizes abstr to functions on the Cartesian product type ($expr \times expr$); it corresponds to biAbstr [50]. It is defined in the same way as abstr, composing dB with its argument and then applying a recursively-defined auxiliary predicate Abstr_2.

Definition 3.35

function Abstr_2 :: $(a\ expr \times a\ expr \Rightarrow a\ dB) \Rightarrow bool$
  Abstr_2 (λ p. CON′ a) = True
  Abstr_2 (λ p. VAR′ n) = True
  Abstr_2 (λ p. S p $$ T p) = (Abstr_2 S $\wedge$ Abstr_2 T)
  Abstr_2 (λ p. ERR′) = True
  Abstr_2 (λ p. BND′ i) = True
  Abstr_2 (λ p. ABS′ (S p)) = Abstr_2 S
  $\neg$ ordinary S $\Longrightarrow$ Abstr_2 S = (S = dB $\circ$ fst $\vee$ S = dB $\circ$ snd)

The predicate Abstr_2 is similar to Abstr, except that it has *two* variable cases: (dB $\circ$ fst) and (dB $\circ$ snd), or equivalently, (λ (x, y). dB x) and (λ (x, y). dB y).

For a concise statement of the main result, we define some abbreviations:

Definition 3.36

abstr_x :: $(a\ expr \times a\ expr \Rightarrow a\ expr) \Rightarrow bool$
  abstr_x S $\equiv \forall$ y. abstr (λ x. S (x, y))
abstr_y :: $(a\ expr \times a\ expr \Rightarrow a\ expr) \Rightarrow bool$
  abstr_y S $\equiv \forall$ x. abstr (λ y. S (x, y))

The predicate abstr_x fixes the second argument y of a two-argument function, and requires the resulting function of x to satisfy abstr for all y :: *expr*. The predicate abstr_y is similar, fixing x and requiring the result to satisfy abstr as a function of y.

Lemma 3.37 (abstr_2 is componentwise abstr)

  abstr_2 S = (abstr_x S $\wedge$ abstr_y S)

This lemma shows that if a two-argument function satisfies abstr in each argument for any fixed value of the other argument, then it satisfies abstr_2. (And the converse,

which is easier.) It is fairly straightforward to prove informally, yet its formal proof was long and required several lemmas. The reasoning is similar to the first part of the proof of Lemma 3.60 (one of the lemmas for the proof of adequacy).

Having thus reduced abstr_2 to componentwise abstr, we may now derive the desired simplification rule for the case (abstr (λ x. LAM y. W x y)).

Lemma 3.38

abstr_LAM: $\forall$ x. abstr (λ y. W x y) $\Longrightarrow$
abstr (λ x. LAM y. W x y) = ($\forall$ y. abstr (λ x. W x y))

This lemma provides a compositional rule for proving abstr conditions on functions of the form (λ x. LAM y. W x y), via the reverse direction of the biconditional. Both directions are also used in the proof of adequacy. It was proved with the help of Lemmas 3.29 (abstr_cases) and 3.37.[10]

We briefly revisit the example term (LAM x. LAM y. x $$ y) from Section 3.2.5, illustrating abstr_LAM by proving that the argument of the outer LAM satisfies abstr, without the use of de Bruijn syntax:

$\forall$ x. abstr (λ y. x)	(by abstr_const)
abstr (λ y. y)	(by abstr_id)
$\forall$ x. abstr (λ y. (x $$ y))	(by abstr_APP)
abstr (λ x. x)	(by abstr_id)
$\forall$ y. abstr (λ x. y)	(by abstr_const)
$\forall$ y. abstr (λ x. (x $$ y))	(by abstr_APP)
abstr (λ x. LAM y. (x $$ y))	(by abstr_LAM)

Not only does the lemma abstr_LAM allow abstr statements to be proved without the use of de Bruijn syntax, but it also completes the task of characterizing *expr* on its own terms – that is, without reference to the underlying de Bruijn syntax. This is demonstrated by the fact that Theorem 3.56 (Adequacy) follows from Hybrid's

[10] Actually a variant of Lemma 3.37 for arbitrary levels (i.e., for Abstr instead of abstr) is used.

lemmas concerning the type *expr*, and it is a significant improvement over both previous versions of Hybrid [2, 18, 51].

We also obtain the characterization of abstr stated in Section 3.1 as a corollary of abstr_LAM:

Lemma 3.39

expand_abstr:

$$\begin{aligned}
&\mathsf{abstr}\ \mathrm{Y} = \big(\mathrm{Y} = (\lambda\ \mathrm{x.}\ \mathrm{x}) \vee \\
&\qquad (\exists\ \mathrm{a.}\ \mathrm{Y} = (\lambda\ \mathrm{x.}\ \mathsf{CON}\ \mathrm{a})) \vee \\
&\qquad (\exists\ \mathrm{n.}\ \mathrm{Y} = (\lambda\ \mathrm{x.}\ \mathsf{VAR}\ \mathrm{n})) \vee \\
&\qquad (\exists\ \mathrm{S\ T.}\ \mathsf{abstr}\ \mathrm{S} \wedge \mathsf{abstr}\ \mathrm{T} \wedge \\
&\qquad\qquad \mathrm{Y} = (\lambda\ \mathrm{x.}\ \mathrm{S\ x\ \$\$\ T\ x})) \vee \\
&\qquad (\exists\ \mathrm{W.}\ (\forall\ \mathrm{x.}\ \mathsf{abstr}\ (\lambda\ \mathrm{y.}\ \mathrm{W\ x\ y})) \wedge (\forall\ \mathrm{y.}\ \mathsf{abstr}\ (\lambda\ \mathrm{x.}\ \mathrm{W\ x\ y})) \wedge \\
&\qquad\qquad \mathrm{Y} = (\lambda\ \mathrm{x.}\ \mathsf{LAM}\ \mathrm{y.}\ \mathrm{W\ x\ y})) \vee \\
&\qquad \mathrm{Y} = (\lambda\ \mathrm{x.}\ \mathsf{ERR})\big)
\end{aligned}$$

The forward implication is equivalent to Lemma 3.29 (abstr_cases), while the reverse implication follows from Lemmas 3.13 and 3.38.

For notational convenience, we extend the abbreviations abstr_x and abstr_y to curried functions, and define another abbreviation abstr_LAM:

Definition 3.40

$$\begin{aligned}
&\mathsf{abstr_x} :: \big([a\ expr,\ a\ expr] \Rightarrow a\ expr\big) \Rightarrow bool \\
&\quad \mathsf{abstr_x}\ \mathrm{W} \equiv \forall\ \mathrm{y.}\ \mathsf{abstr}\ (\lambda\ \mathrm{x.}\ \mathrm{W\ x\ y}) \\
&\mathsf{abstr_y} :: \big([a\ expr,\ a\ expr] \Rightarrow a\ expr\big) \Rightarrow bool \\
&\quad \mathsf{abstr_y}\ \mathrm{W} \equiv \forall\ \mathrm{x.}\ \mathsf{abstr}\ (\lambda\ \mathrm{y.}\ \mathrm{W\ x\ y}) \\
&\mathsf{abstr_LAM} :: \big([a\ expr,\ a\ expr] \Rightarrow a\ expr\big) \Rightarrow bool \\
&\quad \mathsf{abstr_LAM}\ \mathrm{W} \equiv \mathsf{abstr}\ (\lambda\ \mathrm{x.}\ \mathsf{LAM}\ \mathrm{y.}\ \mathrm{W\ x\ y})
\end{aligned}$$

We may then restate Lemma 3.38 (abstr_LAM) as

$$(\mathsf{abstr\_y}\ \mathrm{W} \longrightarrow (\mathsf{abstr\_LAM}\ \mathrm{W} = \mathsf{abstr\_x}\ \mathrm{W})).$$

This lemma does not give a necessary and sufficient condition for (abstr_LAM W). The conjunction (abstr_x W $\wedge$ abstr_y W) is sufficient, but not necessary; while the

implication (abstr_y W $\longrightarrow$ abstr_x W) is necessary, but not sufficient. On its own, (abstr_x W) is neither necessary nor sufficient.

Indeed, when (abstr_y W) fails, there are examples exhibiting all four possible combinations of truth values for (abstr_x W) and (abstr_LAM W):

T	T	$W_1 = (\lambda\ x\ y.\ \mathsf{if}\ y = t_1\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)$
T	F	$W_2 = (\lambda\ x\ y.\ \mathsf{if}\ y = t_1\ \mathsf{then}\ t_1\ \mathsf{else}\ x)$
F	T	$W_3 = (\lambda\ x\ y.\ \mathsf{if}\ y = x\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)$
F	F	$W_4 = (\lambda\ x\ y.\ \mathsf{if}\ y = x\ \mathsf{then}\ t_1\ \mathsf{else}\ x)$

for any fixed t_1, t_2 :: *expr* with $t_1 \neq t_2$.

Fortunately, since Hybrid is intended for reasoning about syntactic terms (for which all subterms are also syntactic), we do not need to handle the case where (abstr_y W) fails. We may assume that an abstr_LAM goal is to be proved via abstr_x and abstr_y; and when given a fact of the form (abstr_LAM W), we may demand a proof of (abstr_y W) to deduce (abstr_x W). This is accomplished using Isabelle's classical reasoner, by declaring safe introduction and elimination rules derived from[11] Lemma 3.38 (abstr_LAM):

Lemma 3.41

abstr_LAM_I: $[\![\, \mathsf{abstr\_x}\ W;\ \mathsf{abstr\_y}\ W \,]\!] \Longrightarrow \mathsf{abstr\_LAM}\ W$
abstr_LAM_E: $[\![\, \mathsf{abstr\_LAM}\ W;\ \mathsf{abstr\_y}\ W;\ (\mathsf{abstr\_x}\ W \Longrightarrow P) \,]\!] \Longrightarrow P$

The lemma abstr_LAM is also declared as a simplifier rule, together with the three rules from Lemma 3.13; but unlike the latter, abstr_LAM is a *conditional* rewrite rule. This means that (abstr_LAM W) will be rewritten to (abstr_x W), wherever it occurs in the goal or premises, but only if (abstr_y W) simplifies to True. Such a rule is usually sufficient for proving abstr subgoals, but it is sometimes unable to simplify such subgoals when they cannot be proved immediately.

[11] In fact abstr_LAM is proved via Lemma 3.41 to simplify the formalization.

Neither the introduction rule from Lemma 3.41 (abstr_LAM_I), nor Lemma 3.38 (abstr_LAM) as a conditional rewrite rule, is efficient in proving abstr goals for functions with nested LAM. The number of subgoals grows exponentially rather than linearly with nesting depth, with many duplicate subgoals. This is not a significant problem for the intended application of Hybrid, namely formalization of the metatheory of object languages. However, the alternative approach considered in Section 3.3 avoids this inefficiency.

The original version of Hybrid [2] provided a special-purpose tactic called abstr_tac for proving abstr subgoals. This was retained for backwards compatibility in the following version [18, 51], but since some changes to object-language theories based on the original version of Hybrid would be required in any case (e.g., to remove proper conditions), abstr_tac has been dropped from the present version in favour of the general-purpose **simp** and **auto**.

While we have characterized the predicate abstr well enough, it might reasonably be asked why we do not use Isabelle/HOL's **typedef** mechanism to define a type *abstr*, as we did for *expr* (which was a predicate proper in [2]).

The trouble with that idea is, how would we construct terms of that type? We could define operators corresponding to the cases of Lemma 3.39 (expand_abstr), but then we would not be using Isabelle/HOL variables to represent object-language variables, which would mean giving up many of the benefits of HOAS. We could instead use the **typedef**-generated bijection applied to a syntactic function, but for partially-specified functions, we would still have to carry along abstr conditions to ensure that the result is well-defined!

The problem is that Isabelle/HOL's type-checking of λ-abstractions does not distinguish syntactic from non-syntactic functions, so in one way or another, this distinction must be handled at the logical level instead.

This idea might be more feasible in another system. For instance, in Coq [6], we could use *dependent pairs* consisting of a function together with a proof that it is

syntactic. Another possibility would be to extend the type system of Isabelle/HOL in such a way as to be able to distinguish syntactic from non-syntactic functions, an idea that has been explored by Howe [36, 37].

3.3 A theory of n-ary syntactic functions

In this section we revisit the question of generalizing abstr to n-ary functions, and develop a different approach from that of Section 3.2.9. Specifically, we consider the use of a single type to represent n-ary functions, for arbitrary n. This was proposed as future work in [50], but to date we know of no published work in that direction[12].

We have seen two examples where the LAM case is handled in a less-than-ideal way:

- In expr_VAR_induct (Lemma 3.33), the induction hypothesis when y = (LAM S) is a property of (S (VAR n)) for all n :: *var*, rather than a property of S :: *expr* ⇒ *expr* directly.

- In proving (abstr (λ x. S x)), when (S x) = (LAM y. W x y), abstr_LAM_I (Lemma 3.41) requires W to satisfy abstr in *both* of its arguments. For nested LAM, the number of subgoals grows exponentially with the nesting depth and many of them are duplicates, as mentioned in Section 3.2.9.

We develop a theory of n-ary syntactic functions that offers solutions to these problems. (We omit many techical details, which are mostly similar to Hybrid itself.) This work is experimental and integration into Hybrid remains as future work. The corresponding theory file is found in Appendix A.

Our representation of n-ary functions on the type *expr* will consist of functions of the type $((nat \Rightarrow expr) \Rightarrow expr)$, where the i^{th} argument will be represented

[12]However, similar techniques have been used in other systems, such as McDowell and Miller's explicit eigenvariable encoding in FO$\lambda^{\Delta\mathbb{N}}$ ([46], section 4.4) which uses functions with list arguments.

as (λ v. v i). This type appears to represent *infinitary* functions; however, those satisfying our n-ary generalization of abstr will provably make use of only finitely many of their arguments.

There are many other possible representations for n-ary functions, but this representation benefits from Isabelle/HOL's support for well-known operations on the function type ($nat \Rightarrow expr$); for instance, composition and functional update can be used for variable renaming and substitution.

We define some type abbreviations for notational convenience:

Definition 3.42

types

$ind = nat$
$a\ iexp = (ind \Rightarrow a\ expr)$
$a\ nexp = (a\ iexp \Rightarrow a\ expr)$
$a\ ndB = (a\ iexp \Rightarrow a\ dB)$

The type *ind* is a synonym for *nat* that will be used to index the arguments of n-ary functions; in effect, it serves to identify variables, much like *bnd* and *var* in Definition 3.1. The type *iexp* is the argument type for our representation of n-ary functions, and *nexp* is the type of the functions themselves. The type *ndB* allows dangling indices, for recursion and induction on de Bruijn syntax.

We next define notation for terms of the types *ndB* and *nexp*:

Definition 3.43 (Operators for *ndB*)

CONn′ a ≡ (λ v. CON′ a)
VARn′ n ≡ (λ v. VAR′ n)
APPn′ S T ≡ (λ v. S v $$′ T v)
ERRn′ ≡ (λ v. ERR′)
BNDn′ j ≡ (λ v. BND′ j)
ABSn′ S ≡ (λ v. ABS′ (S v))
INDn′ i ≡ (λ v. dB (v i))

The first six operators apply the constructors of *dB* pointwise. The last, (INDn′ i), stands for the *n*-ary function whose value is its i$^{\text{th}}$ argument (converted from *expr* to *dB*).

Definition 3.44 (Operators for *nexp*)

CONn a ≡ (λ v. CON a)
VARn n ≡ (λ v. VAR n)
APPn S T ≡ (λ v. S v $$ T v)
ERRn ≡ (λ v. ERR)
INDn i ≡ (λ v. v i)

We define corresponding operators for *nexp*, except for BNDn′ and ABSn′ which will correspond to LAMn when the latter is defined. We may convert between *nexp* and *ndB* by postcomposing with dB or expr; we do not give names to these operations, however we do prove simplifier rules analogous to Lemmas 3.6 and 3.7 (whose statements are omitted here).

Definition 3.45

abstr_n :: *a nexp* ⇒ *bool*
  abstr_n S ≡ Abstr_n (dB ∘ S)
function Abstr_n :: *a ndB* ⇒ *bool*
  Abstr_n (CONn′ a) = True
  Abstr_n (VARn′ n) = True
  Abstr_n (APPn′ S T) = (Abstr_n S ∧ Abstr_n T)
  Abstr_n (ERRn′) = True
  Abstr_n (BNDn′ j) = True
  Abstr_n (ABSn′ S) = Abstr_n S
  ¬ ordinary S ⟹ Abstr_n S = (∃ i. S = INDn′ i)

We define the predicate abstr_n, corresponding to abstr for the case of *n*-ary functions, in a manner completely analogous to Definitions 3.10 and 3.11. The only significant difference is that we now have a natural-number-indexed variable case, (INDn′ i).

Lemma 3.46

Abstr_n_const: Abstr_n (λ v. s)
abstr_n_INDn: abstr_n (INDn i)
abstr_n_const: abstr_n (λ v. s)
abstr_n_APPn: $[\![$ abstr_n S; abstr_n T $]\!] \Longrightarrow$ abstr_n (APPn S T)

These lemmas correspond to Lemmas 3.12 and 3.13, and serve the same purpose.

Definition 3.47

LAMn :: $[\, ind,\ a\ nexp \,] \Rightarrow a\ nexp$
  LAMn i S $\equiv$ expr $\circ$ (Lambda_n i (dB $\circ$ S))
Lambda_n :: $[\, ind,\ a\ ndB \,] \Rightarrow a\ ndB$
  Lambda_n i S $\equiv$ if (Abstr_n S) then (ABSn′ (Lbind_n i 0 S)) else ERRn′

The variable-binding operator LAMn for n-ary functions must take an additional argument i of type *ind* to specify which variable (in the form (INDn i)) to bind. Unlike LAM, the result is again an n-ary function of the same type *nexp*, in which the i$^{\text{th}}$ argument is still present but unused. LAMn is defined in terms of auxiliary functions Lambda_n and Lbind_n, analogous to those of Definitions 3.14 and 3.15.

Definition 3.48

function Lbind_n :: $[\, ind,\ bnd,\ a\ ndB \,] \Rightarrow a\ ndB$
  Lbind_n i j (CONn′ a) = CONn′ a
  Lbind_n i j (VARn′ n) = VARn′ n
  Lbind_n i j (APPn′ S T) = APPn′ (Lbind_n i j S) (Lbind_n i j T)
  Lbind_n i j ERRn′ = ERRn′
  Lbind_n i j (BNDn′ k) = BNDn′ k
  Lbind_n i j (ABSn′ S) = ABSn′ (Lbind_n i (j + 1) S)
  $\neg$ ordinary S $\Longrightarrow$ Lbind_n i j S = (if S = INDn′ i then BND j else S)

The function (Lbind_n i j) replaces occurrences of (INDn′ i) with (BNDn′ j), incrementing j as it proceeds recursively under ABSn′ nodes.

Lemma 3.49

abstr_n_LAMn: abstr_n S $\Longrightarrow$ abstr_n (LAMn i S)

The lemma abstr_n_LAMn is used to compositionally prove abstr_n conditions for terms of the form (LAMn i S) :: *nexp*. Unlike Hybrid as presented in Section 3.2, there is no proliferation of subgoals even with nested LAMn.

Lemma 3.50 (Injectivity for LAMn)

$$[\![\, \mathsf{LAMn}\ i\ S = \mathsf{LAMn}\ i\ T;\ \mathsf{abstr\_n}\ S \vee \mathsf{abstr\_n}\ T \,]\!] \Longrightarrow S = T$$

This lemma corresponds to Theorem 3.21, and its proof is similar.

Theorem 3.51 (Induction for *nexp*)

$$\begin{aligned}
&[\![\, \mathsf{abstr_n}\ U;\ \bigwedge i.\ P\ (\mathsf{INDn}\ i);\\
&\ \bigwedge a.\ P\ (\mathsf{CONn}\ a);\ \bigwedge n.\ P\ (\mathsf{VARn}\ n);\\
&\ \bigwedge S\ T.\ [\![P\ S;\ P\ T]\!] \Longrightarrow P\ (\mathsf{APPn}\ S\ T);\\
&\ \bigwedge i\ S.\ P\ S \Longrightarrow P\ (\mathsf{LAMn}\ i\ S);\ P\ \mathsf{ERRn} \,]\!] \Longrightarrow P\ (U :: \mathit{nexp})
\end{aligned}$$

Our main result in this section is an induction principle for *nexp* that handles the LAMn case directly, unlike the LAM case of expr_VAR_induct (Lemma 3.33).

This induction principle represents open terms using INDn for the free variables. The numeric argument of INDn functions like a variable name, and to use this form of induction we would need to keep track of free variables. The technical lemmas to support this have not yet been developed, but they should be similar to those used for VAR in the original version of Hybrid [2].

On the other hand, we may still use function application to perform substitution: to substitute S :: *nexp* for (INDn i) in T :: *nexp*, we use functional update notation to define

$$\mathsf{subst\_nexp}\ i\ S\ T \equiv \big(\lambda\ v.\ S\ (v\ (i := T))\big).$$

This method of performing substitution generalizes easily to *simultaneous* substitution, which is useful for applications such as strong normalization proofs (as noted by Urban [73]).

3.4 Adequacy

Adequacy is the term traditionally used for properties of a HOAS representation that are meant to show its correctness, typically by establishing some form of equivalence with ordinary named-variable syntax.

In the setting of logical frameworks [32, 58] (and also in proof assistants based on constructive type theory such as Coq), there is a notion of *definitional equality* generated by a confluent and strongly normalizing reduction relation, such as $\beta\eta$-reduction in a typed λ-calculus. The η-long β-normal forms are called *canonical forms*, and every term is definitionally equal to a unique canonical form.[13] Adequacy consists of establishing a compositional bijection between the canonical forms of certain types and the object-logic terms to be represented.

The classical higher-order logic of Isabelle/HOL, however, does not fit into that approach. Rather than definitional equality, we have *provable equality* which does not satisfy such nice properties. In particular, as a corollary of Gödel's first incompleteness theorem, any type with at least 2 elements includes terms that are not provably equal to any of the concrete elements of that type. So we will need a different notion of adequacy; there are several candidates:

- We could work with a restricted sublanguage of Isabelle/HOL, which behaves as a logical framework with canonical forms, as done by Felty and Momigliano [21]. A similar option is to model Hybrid as a theory in a logical framework, defined separately from Hybrid itself, which is the approach used by Crole [13] to prove adequacy for the previous version of Hybrid.

  This approach leads to adequacy proofs that are similar to the traditional ones for logical frameworks, yet it is not fully satisfactory because of the lack of a

[13] Some systems (such as LF) have a definitional equality generated by β-reduction only. Canonical forms are still required to be η-long, so there will be terms that are not definitionally equal to any canonical form. This complicates the picture somewhat but it is not relevant for our present purposes.

solid formal correspondence between the system for which adequacy is proved and the actual Isabelle/HOL formal theory of Hybrid.

- We could attempt to show an equivalence between Hybrid and named-variable syntax *within* Isabelle/HOL.

  This approach would give a direct connection with the actual formal theory of Hybrid, and it is straightforward enough for closed terms. (A similar correspondence, using de Bruijn indices in place of named variables, was used in the *definition* of Hybrid.) However, it would require a suitable representation for open terms in Hybrid. This could be VAR, or the n-ary syntactic functions from Section 3.3; but either would complicate the statement and proof of adequacy.

  It is also doubtful whether this approach could be extended to prove adequacy for two-level encodings that use shallow embeddings into Isabelle/HOL for some object-language features, as found in Chapters 6 to 9.

- We could work with a *set-theoretic semantics* for classical higher-order logic [63]. Elements of sets given by a semantic model take the place of the canonical forms in a traditional proof of adequacy.

  This approach provides a precise correspondence with the formal theory of Hybrid while supporting straightforward reasoning on open terms of Isabelle/HOL. It can also borrow elements from either of the other two approaches. It is the approach we take in the present work.

It might seem that working directly with a system as large and complicated as Isabelle/HOL would unduly complicate the proof of adequacy. However, most of the complicated parts of Isabelle/HOL are built definitionally, so they do not impose the burden of proving consistency. Nor is it necessary to unfold complicated definitions, as the necessary properties of the defined constructs are available as formal theorems.

Thus, most of Isabelle/HOL's features, including even those actually used in the development of Hybrid, were not an obstacle at all.

There is one exception: Isabelle/HOL's extensions to simple types, notably type classes and overloading. These features would complicate a detailed presentation of semantics for Isabelle/HOL, such as would be required to prove consistency of its axioms and soundness of its inference rules and definitional mechanisms. We will instead assume these properties without proof, in effect limiting the scope of our work to correctness of Hybrid *within* Isabelle/HOL, as opposed to correctness of Hybrid *and* Isabelle/HOL.

3.4.1 Definitions and Notation

We mainly follow Pitts [63]. However, that is a semantics for the HOL system [35] rather than Isabelle/HOL. For specifics of the latter system, we follow Krauss and Schropp [42], and to some extent Wenzel [75]. The notation and ideas are largely standard, with a few exceptions indicated where they first occur.

We assume that type classes and overloaded constant definitions are translated out in the manner of [42, § 5]. Overloaded constant definitions amount to case distinction on the *syntax* of types,[14] so it is more appropriate to deal with them syntactically (by translation) rather than semantically. Type classes must then be translated out as well, so that the translation of overloading can be applied to their logical content. In the proof of adequacy, we will work only with monomorphic types and terms, which are unchanged by these translations; so we omit the details.

Sequences of finite but arbitrary length occur frequently in definitions of syntax and semantics. For conciseness of exposition, we use the notation $\overline{x_n}$ to mean $x_1, \ldots, x_n$, for various formulas x. Unless otherwise indicated, the length n of the sequence is an arbitrary non-negative integer. (This notation is found, e.g., in [42].)

[14] Also primitive recursion and structural induction, in Isabelle.

Types

Let Ω be a finite set of *type declarations*, which are pairs (ν, n) where ν is a type constant name and $n \in \mathbb{N}$ is its arity. (We require that the first components of these pairs be distinct, i.e., type constants are declared at most once.)

A type constant ν of arity 0 is called a base type, and it is modeled by a set $M(\nu)$. A type constant ν of arity $n > 0$ is called a type constructor, and it is modeled by a mapping $M(\nu)$ from n-tuples of sets to sets. We assume that these sets are elements of a *universe* $\mathcal{U}$, whose properties we will consider later. (This could be either a set or a proper class, though the latter option would require a metalanguage somewhat stronger than ZFC [42, § 3].) Base types and type constructors may be treated uniformly by identifying functions $f : \mathcal{U}^0 \to \mathcal{U}$ with the corresponding sets $f() \in \mathcal{U}$.

We define the syntax of *types* recursively by:

$$\tau ::= \alpha \mid (\overline{\tau_n})\ \nu$$

where τ and τ_i stand for types, α for a type variable, and ν for a type constant of arity n. We will use the usual Isabelle/HOL concrete syntax for types, dropping the parentheses when $n \leq 1$ and using infix notations where convenient. We write $\mathrm{TV}(\tau)$ for the set of type variables occurring in τ. A type τ is called *polymorphic* if $\mathrm{TV}(\tau) \neq \varnothing$, *monomorphic* otherwise.

We assume that Ω contains at least the base type *prop* (propositions) and the type constructor $\Rightarrow$ of arity 2 (function space, written infix). We further assume that $M(\mathit{prop}) = \{0, 1\}$ and $M(\Rightarrow)(Y_1, Y_2)$ is the set of functions from Y_1 to Y_2. Thus, we restrict our attention to *classical, standard* models.

Given a finite sequence of distinct type variables $\overline{\alpha_n}$, and corresponding sets $X_i \in \mathcal{U}$ for $i \in \{1, \ldots, n\}$, we recursively define the *interpretation* $[\![\tau]\!] \in \mathcal{U}$ for types τ

with $\mathrm{TV}(\tau) \subseteq \{\overline{\alpha_n}\}$ by:[15]

$$[\![\alpha_i]\!] = X_i$$
$$[\![(\overline{\tau_m})\ \nu]\!] = M(\nu)\big(\overline{[\![\tau_m]\!]}\big)$$

This induces a function $[\![\overline{\alpha_n}.\tau]\!] : \mathcal{U}^n \to \mathcal{U}$ given by $\big(\overline{X_n}\big) \mapsto [\![\tau]\!]$. We call $\overline{\alpha_n}$ a *type context* and $\overline{\alpha_n}.\tau$ a *type-in-context*. We will always write the context $\overline{\alpha_n}$ and the function arguments $\overline{X_n}$ explicitly, except for monomorphic types (where the context is empty) and in the definition of $[\![\_]\!]$ on terms (where we will again fix a correspondence between type variables and sets).

This approach of interpreting types-in-context by explicit functions on $\mathcal{U}^n$ is from Pitts [63]; a more traditional approach is to interpret types in an *environment* consisting of a function from type variables to $\mathcal{U}$.

A *type substitution* is a function δ from type variables to types. We write $\tau[\delta]$ for the type obtained by replacing each type variable α in τ with $\delta(\alpha)$, i.e., the usual notion of simultaneous substitution. We may also write a type substitution explicitly as $\delta = \overline{\tau_n/\alpha_n}$, which means $\delta(\alpha_i) = \tau_i$ for $i \in \{1, \ldots, n\}$ and $\delta(\alpha) = \alpha$ for $\alpha \notin \{\overline{\alpha_n}\}$.

Terms

Let Σ be a finite set of *constant declarations*, which are pairs (c, τ) where c is a constant name and τ is its type. (We again require that the first components of the pairs be distinct.)

For each constant c of type τ, we define a canonical type context $\mathrm{TC}(c) = \overline{\alpha_n}$ listing the type variables of τ in some fixed order.[16] Then c is modeled by a mapping $M(c)$, giving for each tuple of sets $\big(\overline{X_n}\big) \in \mathcal{U}^n$ an element $M(c)\big(\overline{X_n}\big)$ of $[\![\overline{\alpha_n}.\tau]\!]\big(\overline{X_n}\big)$. When τ is monomorphic, this simplifies to $M(c) \in [\![\tau]\!]$.

[15] The notation $[\![\_]\!]$, sometimes called *Scott brackets*, is quite standard; Isabelle's use of these brackets to group the premises of an implication leads to an unfortunate notational clash here. We will simply avoid stating any Isabelle/HOL propositions in this form in Section 3.4.

[16] This can be done without the use of an ordering on type variables, e.g., by listing them in the order in which they occur in τ.

We then recursively define the syntax of *terms* – a simply-typed λ-calculus with constants:

$$t_\tau ::= x_\tau \mid c_\tau \mid (t_{\tau\Rightarrow\tau'}\ t'_\tau) :: \tau' \mid (\lambda x_\tau.\, t_{\tau'}) :: \tau \Rightarrow \tau'$$

where t, t' stand for terms, x for a variable, c for a constant, and τ, τ' for types. This is a Church-style presentation, in which we define syntax for just the well-typed terms and simultaneously give their types. (We write a term t of type τ as either t_τ or $t :: \tau$.) We will use Isabelle/HOL concrete syntax for terms, including infix notations and omission of parentheses and types that can be reconstructed from context.

A variable x_τ is treated as a pair consisting of a name x and a type τ, so variables with the same name but different types are considered distinct.[17] (A variable and a constant with the same name are also considered distinct, regardless of type.) We then have the standard notions of free and bound variables, and α-equivalence; we write $\mathrm{FV}(t)$ for the set of free variables of a term t. We also write $\mathrm{TV}(t)$ for the set of type variables occurring in a term t, calling t polymorphic if this set is nonempty, or monomorphic otherwise. Note that a term $t :: \tau$ can be polymorphic even if the type τ is monomorphic; an example from [63] is $(f_{\alpha\Rightarrow b}\ x_\alpha) :: b$ where b is a base type.

The constant $c_{\tau'}$ is well-typed if $(c, \tau) \in \Sigma$ and $\tau' = \tau[\delta]$ for some type substitution δ. (This allows specialization of polymorphic constants to type instances.) We will implicitly assume that constants are well-typed. By [63, Lemma 1], the values of δ on the type variables of τ are uniquely determined by τ and τ'.

We assume that Σ contains at least the constants

$$\begin{array}{lll} \Longrightarrow & :: [\mathit{prop}, \mathit{prop}] \Rightarrow \mathit{prop} & \text{(implication)} \\ \equiv & :: [\alpha, \alpha] \Rightarrow \mathit{prop} & \text{(equality)} \\ \bigwedge & :: (\alpha \Rightarrow \mathit{prop}) \Rightarrow \mathit{prop} & \text{(universal quantification)} \end{array}$$

and that they are interpreted as follows:

[17] This is a technical convenience; the actual Isabelle/HOL system uses type inference, and reports a type error if it is unable to give the same type to all occurrences of a variable name.

- $M(\Longrightarrow)(p)(q)$ is 0 if $p = 1$ and $q = 0$, otherwise 1;
- $M(\equiv)(x)(y)$ is 1 if $x = y$, otherwise 0;
- $M(\bigwedge)(f)$ is 1 if the function f is constant with value 1, otherwise 0.

We again fix a type context $\overline{\alpha_n}$ and corresponding sets $X_i \in \mathcal{U}$ for $i \in \{1, \ldots, n\}$, which will later become explicit arguments of the interpretation function. Until then, we implicitly require that all types and terms have only type variables from $\{\overline{\alpha_n}\}$, unless otherwise indicated.

A finite sequence of (term) variables $\overline{x_m}$, not necessarily distinct, is called a *context*; and $\overline{x_m}\,.\,t$ is called a *term-in-context* when t is a term with $\mathrm{FV}(t) \subseteq \{\overline{x_m}\}$.

Given a term-in-context $\overline{x_m}\,.\,t$ with $t :: \tau$ and $[\![\tau]\!] = Y$, and $x_i :: \tau_i$ and $[\![\tau_i]\!] = Y_i$ for $i \in \{1, \ldots, m\}$, we define the interpretation $[\![\overline{x_m}\,.\,t]\!] : Y_1 \times \cdots \times Y_m \to Y$ by recursion on t simultaneously for all contexts $\overline{x_m}$ containing $\mathrm{FV}(t)$:

- If t is a variable, then it must be equal to at least one of the x_i, and there is a unique largest k such that $t = x_k$; we set $[\![\overline{x_m}\,.\,t]\!](\overline{y_m}) = y_k$.
- If $t = c_\tau$ where $(c, \tau') \in \Sigma$ with $\tau = \tau'[\delta]$ and $\mathrm{TC}(c) = \overline{\beta_p}$, then we set

  $$[\![\overline{x_m}\,.\,c_\tau]\!](\overline{y_m}) = M(c)\big(\overline{S_p}\big)$$

  where $S_j = [\![\delta(\beta_j)]\!]$ for $j \in \{1, \ldots, p\}$.

  Note that τ' and $\mathrm{TC}(c) = \overline{\beta_p}$ may have type variables that are not in $\{\overline{\alpha_n}\}$; and the $\delta(\beta_j)$ are uniquely determined by τ and τ', as noted above.

  We have $M(c)\big(\overline{S_p}\big) \in [\![\overline{\beta_p}\,.\,\tau']\!]\,\big(\overline{S_p}\big)$; equality of this set with $[\![\tau]\!] = [\![\tau'[\delta]]\!]$ is a type substitution property [63, Lemma 2] that follows by a straightforward induction on τ' from the definition of $[\![\_]\!]$ for types.
- If $t = (t_1\ t_2)$ where $t_1 :: \tau' \Rightarrow \tau$ and $t_2 :: \tau'$ and $[\![\tau']\!] = Y'$, then

  $$[\![\overline{x_m}\,.\,(t_1\ t_2)]\!](\overline{y_m}) = f(y)$$

  where $f = [\![\overline{x_m}\,.\,t_1]\!](\overline{y_m}) : Y' \to Y$ and $y = [\![\overline{x_m}\,.\,t_2]\!](\overline{y_m}) \in Y'$.

- If $t = (\lambda x.\, t')$ where $x :: \tau'$ and $t' :: \tau''$, then $\tau = (\tau' \Rightarrow \tau'')$ and $Y = [\![\tau' \Rightarrow \tau'']\!]$ is the set of functions from $[\![\tau']\!] = Y'$ to $[\![\tau'']\!] = Y''$; we set

$$[\![\overline{x_m}\,.\,(\lambda x.\, t')]\!](\overline{y_m})(y) = [\![(\overline{x_m}, x)\,.\,t']\!](\overline{y_m}, y)$$

  for all $y \in Y'$.

A *substitution* is a function γ from variables to terms, respecting types. We write $t[\gamma]$ for the term obtained by first renaming the bound variables of t to be disjoint from $\mathrm{FV}(\gamma(x_\tau))$ for all $x_\tau \in \mathrm{FV}(t)$, and then replacing each free variable x_τ in the resulting term t' with $\gamma(x_\tau)$. This is the usual notion of capture-avoiding simultaneous substitution; to obtain a uniquely-defined result, the renaming must be performed deterministically (e.g., with the help of a fixed total order on variables), but we omit the details. We may also write a substitution explicitly as $\gamma = \overline{t_m/x_m}$, with the convention that variables not named are mapped to themselves, just as we did for type substitutions.

By induction on terms, we have a substitution lemma analogous to [63, Lemma 4]: given a term-in-context $\overline{x_m}\,.\,t$, another context $\overline{x'_p}$, and a substitution γ such that $\mathrm{FV}(\gamma(x_i)) \subseteq \{\overline{x'_p}\}$ for $i \in \{1, \ldots, m\}$, we have

$$[\![\overline{x'_p}\,.\,t[\gamma]]\!]\left(\overline{y'_p}\right) = [\![\overline{x_m}\,.\,t]\!](\overline{y_m})$$

where $y_i = [\![\overline{x'_p}\,.\,\gamma(x_i)]\!]\left(\overline{y'_p}\right)$ for $i \in \{1, \ldots, m\}$.

As a corollary, $[\![\overline{x_m}\,.\,\_]\!]$ identifies α-equivalent terms, so by renaming variables as needed, we may arrange for contexts to consist of distinct variables.[18]

We write $[\![\overline{\alpha_n}, \overline{x_m}\,.\,t]\!]$ for the function mapping $\left(\overline{X_n}\right) \in \mathcal{U}^n$ to $[\![\overline{x_m}\,.\,t]\!]$, which is itself a function $Y_1 \times \cdots \times Y_m \to Y$, with $Y = [\![\overline{\alpha_n}\,.\,\tau]\!]\left(\overline{X_n}\right)$ where $t :: \tau$, and $Y_i = [\![\overline{\alpha_n}\,.\,\tau_i]\!]\left(\overline{X_n}\right)$ where $x_i :: \tau_i$ (for $i \in \{1, \ldots, m\}$). Henceforth, the context for a polymorphic term will include a type context, and its interpretation will be a function

[18] Or even variables that differ in their name components, which avoids problems when substituting types for type variables in a term.

taking explicit set arguments $\left(\overline{X_n}\right) \in \mathcal{U}^n$, in this manner. (This is a dependently-typed function, or set-theoretically, an element of a Cartesian product indexed by elements of $\mathcal{U}^n$.) However, for monomorphic terms we may omit this complication, and for a *closed* monomorphic term $t :: \tau$, we will simply have $[\![t]\!] \in [\![\tau]\!]$ where $[\![\tau]\!]$ is a set in $\mathcal{U}$.

We omit the statement of a type substitution lemma analogous to [63, Lemma 3] since we will be working only with monomorphic terms.

Theories and Models

A *theory* T has the form (Ω, Σ, Φ) where Ω and Σ are sets of type and constant declarations respectively, as defined above, and Φ is a finite set of *axioms*, i.e., closed terms of type *prop*. The *theorems* of T are the closed terms of type *prop* derivable from the axioms by Isabelle/HOL's inference rules. These rules take the form of a sequent calculus for higher-order logic, a portion of which is detailed in [42].

A *model* M for a theory T consists of interpretations $M(\nu)$ for each $(\nu, n) \in \Omega$ and $M(c)$ for each $(c, \tau) \in \Sigma$, as defined above, such that $[\![\phi]\!] = 1$ (if monomorphic) or $[\![\phi]\!]$ is a constant function with value 1 (if polymorphic) for each $\phi \in \Phi$. We could write $[\![\_]\!]_M$ to make the dependency on M explicit, but in most cases it will be clear from context.

A theory T is *syntactically consistent* if not all propositions of T are theorems of T; equivalently, if T does not prove the single proposition $(\bigwedge(\mathrm{P} :: \mathit{prop}).\, P)$.

A theory T is *semantically consistent* if it has a model. This is a stronger property than syntactic consistency, because higher-order logic is not complete for standard models such as those considered here.

A theory T' *extends* T if each component of T is a subset of the corresponding component of T'. It is a *definitional extension* if all added type and term constants and axioms result from a sequence of applications of Isabelle/HOL's definition mech-

anisms. We omit the details of the latter, but they are similar to Pitts' constant and type definitions [63, § 16.5].

3.4.2 Assumptions

In the following sections, we consider an Isabelle/HOL theory T that definitionally extends Hybrid and has a type *con* that we will use to instantiate Hybrid's type parameter for constants. (The case study presented in Chapters 5 to 10 provides examples of such theories.) This instantiation will generally be left implicit, e.g., (*con expr*) will be abbreviated to *expr*. We will also implicitly require that all types and terms considered are monomorphic.

We assume semantic consistency of T, which follows from semantic consistency of a much smaller theory consisting of Isabelle/HOL's primitive constants and axioms, together with preservation of semantic consistency by Isabelle/HOL's definition mechanisms. We let M be an arbitrary model of T, which will be used implicitly to interpret types and terms.

We assume soundness of Isabelle/HOL's inference rules, so that $[\![\phi]\!] = 1$ (if monomorphic) or $[\![\phi]\!]$ is a constant function with value 1 (if polymorphic) for all theorems ϕ of T, rather than just for its axioms.

We assume that $[\![bool]\!]$ is bijective with $[\![prop]\!]$, and we identify those two sets. We also assume that the logical connectives of Isabelle/HOL have their usual meaning. (These assumptions are likely to be implied by Isabelle/HOL's axioms, and easily proven with the help of its formal lemmas, given a classical standard model for *prop* and its connectives as we have assumed.)

While we do not assume that the universe $\mathcal{U}$ is the class of all sets, we will assume that particular sets such as the natural numbers are elements of $\mathcal{U}$ where convenient. We will also assume closure properties as needed for semantic consistency, as in [63]; it is enough that $\mathcal{U}$ contains an infinite set, all non-empty subsets of sets in $\mathcal{U}$, and

all sets of functions Y^X for $X, Y \in \mathcal{U}$.

Soundness also requires one property of $\mathcal{U}$: that its elements are all non-empty sets. (Unlike some other systems such as Coq, Isabelle/HOL builds in the assumption that types are non-empty.) We will also require the Axiom of Choice to obtain elements of the interpretations of polymorphic types, and if $\mathcal{U}$ were a proper class, we would need a stronger form of it such as global choice (as in [42]). However, ZFC set theory contains sets that meet all of our requirements for $\mathcal{U}$.[19]

In practice, these assumptions will allow us to use Isabelle/HOL formal theorems as mathematical statements about models of the theory in a straightforward manner. In particular, we may apply formal induction principles not only to formally defined predicates, but also to arbitrary mathematical properties. We will use formal theorems provided as part of Isabelle/HOL as well as those stated in Section 3.2 (and proved in the corresponding theory file [45]).

Since most of Isabelle/HOL is built definitionally, it would not be unreasonably difficult to prove the consistency and soundness properties that we are assuming without proof. However, features such as type classes and overloaded definitions would complicate such a proof considerably (e.g., as compared with Pitts' consistency proofs in [63]), and we view it as outside the scope of the present work.

Since our notion of model uses the full set-theoretic function space, we do not have a completeness property; that would require Henkin models [33]. This is a well-known limitation of HOL with standard models, and it does not cause problems for adequacy. (Indeed, adequacy would not hold for more general models.[20])

In addition to the general assumptions above, we require one technical assumption related specifically to Hybrid. We assume the existence of a set L_{con} of closed

[19] Such a set can be constructed by transfinite recursion as in the von Neumann cumulative hierarchy, as mentioned in [63].

[20] As explained by Henkin [33], it is a corollary of completeness (via compactness) that no infinite structure can be axiomatized categorically on general models. (This is the same situation as for first-order logic.)

T-terms of type *con* such that for any model M of T, the function $[\![\_]\!]_M$ restricted to L_{con} is a bijection onto $[\![con]\!]_M$. This is in effect a base case for adequacy of Hybrid. It will hold in the common case where *con* is a datatype with only 0-ary constructors, or with constructors whose argument types satisfy the same property.

However, it does not hold for all types: for instance, the set of all functions from any infinite set to any set with at least two elements is uncountable, and thus cannot be in bijection with any set of terms. This is the main reason why we use the type (*con expr*) rather than the polymorphic type (*a expr*) in the proof of adequacy. These issues will be discussed further in Section 3.4.4.

A similar "base case" is needed also for the type *nat*, but this one is a consequence of the assumption of soundness:

Proposition 3.52 *The function from* $\mathbb{N}$ *to* $[\![nat]\!]$ *defined by* $n \mapsto [\![\mathsf{Suc}^n\ 0]\!]$*, where* $(\mathsf{Suc}^n\ 0)$ *is the term consisting of* n *applications of* Suc *to* 0*, is a bijection.*

This follows from Isabelle/HOL's basic theorems for *nat* (namely Suc_not_Zero, inj_Suc, and nat_induct). Indeed, those theorems, together with the typing rules for 0 and Suc and Isabelle/HOL's basic theorems for equality, are precisely the Peano axioms for natural number arithmetic (with second-order induction), which are well-known to characterize the natural numbers uniquely in a standard model.

For notational convenience, we will identify $[\![nat]\!]$ with $\mathbb{N}$ and $[\![\mathsf{Suc}^n\ 0]\!]$ with n. We will also define a set $L_{nat} = \{\mathsf{Suc}^n\ 0 \mid n \in \mathbb{N}\}$ of closed terms of type *nat*, called the *numerals*.

3.4.3 Proof of Adequacy

Crole's adequacy result [13] is based on a bijection between a lambda calculus (representing Hybrid's syntax with LAM) and canonical forms of a logical framework (in a signature corresponding to our type *dB*), both of which had to be defined for that

purpose. We will use the actual Isabelle/HOL syntax of the type *expr* and its operators in place of the lambda calculus, and elements of sets given by the model M in place of the canonical forms.

Definition 3.53 *Let $\mathcal{LE}$ be the smallest set of terms of type expr containing* ERR*,* (CON c) *for all $c \in L_{con}$,* (VAR n) *for all $n \in L_{nat}$,* (APP s_1 s_2) *for all $s_1, s_2 \in \mathcal{LE}$, and x and* (LAM $x.\, s$) *for all variables x of type expr and all $s \in \mathcal{LE}$.*

This set $\mathcal{LE}$ of "lambda-expressions" includes open terms of type *expr*; it makes precise the notion of syntactic terms from Section 3.1. Here we diverge significantly from Crole [13], as we do not translate the free variables of such terms into the form (VAR n); we instead use the semantics directly, with its interpretation of open terms as functions from a Cartesian power $[\![expr]\!]^n$ to $[\![expr]\!]$.

Definition 3.54 *Let M^n_{expr} be the set of all functions $f\colon [\![expr]\!]^n \to [\![expr]\!]$ such that for any function $g\colon [\![expr]\!] \to [\![expr]\!]^n$, if $\pi_i \circ g$ is the identity function for some i and $\pi_j \circ g$ is a constant function for all $j \neq i$, then $[\![\mathsf{abstr}]\!](f \circ g) = 1$.*

For $n = 0$, the condition is vacuous and the domain of the functions is a one-element set, so we may identify M^0_{expr} with $[\![expr]\!]$. For $n = 1$, M^1_{expr} is the set of functions $f\colon [\![expr]\!] \to [\![expr]\!]$ satisfying $[\![\mathsf{abstr}]\!](f) = 1$. For $n \geq 2$, the set M^n_{expr} corresponds to an n-ary generalization of abstr. It is possible to define such a predicate in Isabelle/HOL for any fixed n, e.g., abstr_2 from Section 3.2.9 (based on Momigliano et al.'s biAbstr [50]) for $n = 2$. However, we cannot generalize this to a predicate with n as an argument, due to the lack of dependent types in Isabelle/HOL. (Ways of working around this limitation were discussed in Section 3.2.9.)

Definition 3.55 *Given a context $\sigma = \overline{x_n}$ consisting of distinct variables of type expr, define L^σ_{expr} to be the set of terms in $\mathcal{LE}$ with free variables from σ.*

We will henceforth implicitly require that a "context σ" consists of distinct variables of type *expr*. For any term $t \in L^{\sigma}_{expr}$, $\sigma\,.\,t$ is a term-in-context; and if n is the length of σ, then $[\![\sigma\,.\,t]\!]$ is a function from $[\![expr]\!]^n$ to $[\![expr]\!]$.

Theorem 3.56 (Adequacy of Hybrid) *For any context $\sigma = \overline{x_n}$, there is a bijection $\phi_\sigma : L^{\sigma}_{expr}/{\sim_\alpha} \to M^n_{expr}$ given by $\phi_\sigma\big([t]_\alpha\big) = [\![\sigma\,.\,t]\!]$.*

Furthermore, these bijections are compositional, in the sense that for any contexts $\sigma = \overline{x_n}$ and $\sigma' = \overline{x'_m}$ and any substitution γ with $\gamma(x_i) \in L^{\sigma'}_{expr}$ for $i \in \{1, \ldots, n\}$,

$$\phi_{\sigma'}\big([t[\gamma]]_\alpha\big)(\overline{y_m}) = \phi_\sigma\big([t]_\alpha\big)\big(\overline{y'_n}\big)$$

for all $(\overline{y_m}) \in [\![expr]\!]^n$, where $y'_i = \phi_{\sigma'}\big([\gamma(x_i)]_\alpha\big)(\overline{y_m})$ for $i \in \{1, \ldots, n\}$.

This theorem will be proved with the help of several lemmas.

Lemma 3.57 *The function $t \mapsto [\![\sigma\,.\,t]\!]$ is well-defined on alpha-equivalence classes.*

This is a corollary of [63, Lemma 4], as described in Section 3.4.1.

Lemma 3.58 *If $t \in L^{\sigma}_{expr}$, then $[\![\sigma\,.\,t]\!] \in M^n_{expr}$.*

Proof. We show that $[\![\sigma_{\hat{x}}\,.\,(\mathsf{abstr}\ (\lambda x.\,t))]\!](y) = 1$ for all $x \in \sigma$ and $y \in [\![expr]\!]^{n-1}$, where $\sigma_{\hat{x}}$ is σ with x removed. The proof is by induction on t with cases from Definition 3.53, using Isabelle/HOL theorems from Lemma 3.13:

- If t is a variable, then the claim follows either from abstr_id if $t = x$, or from abstr_const otherwise.
- If $t = \mathsf{ERR}$, $t = (\mathsf{CON}\ c)$, or $t = (\mathsf{VAR}\ n)$, then the claim follows from abstr_const.
- If $t = (\mathsf{APP}\ s_1\ s_2)$, then the claim follows from the induction hypothesis by abstr_APP.

- If $t = (\mathsf{LAM}\ x'.\, s)$, then the claim follows either from abstr_const if $x' = x$, or from the induction hypothesis by abstr_LAM (Lemma 3.38) otherwise.

Then by the definition of $[\![\_]\!]$ for terms, we have

$$[\![\mathsf{abstr}]\!]\big([\![\sigma_{\hat{x}}\,.\,(\lambda x.\, t)]\!](y)\big) = 1$$
$$[\![\mathsf{abstr}]\!]\big(z \mapsto [\![(\sigma_{\hat{x}}, x)\,.\,t]\!](y, z)\big) = 1$$
$$[\![\mathsf{abstr}]\!]\big([\![\sigma\,.\,t]\!] \circ g\big) = 1$$

where in the last step, we have permuted the context and g is the function sending $z \in [\![expr]\!]$ to y with z inserted in the position where x occurs in σ. But this is an arbitrary function of the form specified in Definition 3.54, so $[\![\sigma\,.\,t]\!] \in M^n_{expr}$. □

Lemma 3.59 *If $t_1, t_2 \in L^\sigma_{expr}$ and $[\![\sigma\,.\,t_1]\!] = [\![\sigma\,.\,t_2]\!]$, then $t_1 \sim_\alpha t_2$.*

Proof. We proceed by induction on t_1. If either t_1 or t_2 is a variable, then either they are the same variable and the conclusion is immediate, or there is a substitution γ of closed terms for the variables in σ that takes t_1 and t_2 to different cases of Definition 3.53. By the term substitution property for $[\![\_]\!]$, the premise of the lemma implies $[\![t_1[\gamma]]\!] = [\![t_2[\gamma]]\!]$, but this contradicts expr_distinct (Lemma 3.22). So we may assume that neither t_1 nor t_2 is a variable; and again by expr_distinct, they must fall into the same case of Definition 3.53:

- If $t_1 = (\mathsf{CON}\ c_1)$ and $t_2 = (\mathsf{CON}\ c_2)$, then the premise $[\![\sigma\,.\,t_1]\!] = [\![\sigma\,.\,t_2]\!]$ implies $[\![c_1]\!] = [\![c_2]\!]$ by expr_inject (Lemma 3.23), and thus $c_1 = c_2$ by the assumed property of L_{con} (in Section 3.4.2), so we have $t_1 \sim_\alpha t_2$ as required (indeed they are equal).

  The VAR case is similar, while the ERR case is trivial.

- If $t_1 = (\mathsf{APP}\ s_1\ s_2)$ and $t_2 = (\mathsf{APP}\ s_3\ s_4)$, then we have $[\![\sigma\,.\,s_1]\!] = [\![\sigma\,.\,s_3]\!]$ and $[\![\sigma\,.\,s_2]\!] = [\![\sigma\,.\,s_4]\!]$ by expr_inject, and thus $s_1 \sim_\alpha s_3$ and $s_2 \sim_\alpha s_4$ by the induction hypothesis. Together, these imply $t_1 \sim_\alpha t_2$.

- If $t_1 = (\mathsf{LAM}\ x_1.\, s_1)$ and $t_2 = (\mathsf{LAM}\ x_2.\, s_2)$, choose a variable x' of type *expr* that does not occur in σ, and define $s_i' = s_i[x'/x_i]$ and $t_i' = (\mathsf{LAM}\ x'.\, s_i')$; by definition, we have $t_i \sim_\alpha t_i'$. Since the semantics equates α-variants, the premise implies $[\![\sigma\,.\,t_1']\!] = [\![\sigma\,.\,t_2']\!]$ which in turn implies $[\![\sigma\,.\,(\lambda\ x'.\, s_1')]\!] = [\![\sigma\,.\,(\lambda\ x'.\, s_2')]\!]$ by expr_inject. (The abstr premise of expr_inject is provided by Lemma 3.58 via Definition 3.54.) From the semantics of λ, this equality is equivalent to $[\![(\sigma, x')\,.\,s_1']\!] = [\![(\sigma, x')\,.\,s_2']\!]$ and we may apply the induction hypothesis to conclude that $s_1' \sim_\alpha s_2'$. Thus we have $t_1' \sim_\alpha t_2'$ and finally $t_1 \sim_\alpha t_2$ as required. □

Lemma 3.60 *For any context $\sigma = \overline{x_n}$, if $f \in M^n_{expr}$, then there is a term $t \in L^\sigma_{expr}$ such that $[\![\sigma\,.\,t]\!] = f$.*

Proof. Let $y_0 = [\![\mathsf{ERR}]\!]$; we have $[\![\mathsf{size}]\!](y_0) = 0$ by size_ERR (Lemma 3.31). We proceed by induction on $[\![\mathsf{size}]\!](f(y_0, y_0, \ldots, y_0))$.

Suppose $f \circ g$ is the identity function on $[\![expr]\!]$, for some $g\colon [\![expr]\!] \to [\![expr]\!]^n$ with $\pi_i \circ g = \mathrm{id}$ and all other projections of g being constant functions. We claim that in this case $f = \pi_i$. This is equivalent to $f \circ g' = \mathrm{id}$ for *all* such functions g' (for the same i).

Suppose not; then there are functions g_1, g_2 satisfying the same conditions as g above, such that $f \circ g_1 = \mathrm{id}$ and $f \circ g_2 \neq \mathrm{id}$. We may assume, without loss of generality, that g_1 and g_2 differ only in the j^{th} component for some $j \neq i$; let y_1 and y_2 be their respective constant values on this component. By expand_abstr (Lemma 3.39), $f \circ g_2$ must take values corresponding to a single case of Definition 3.53. Choose $y \in [\![expr]\!]$ from a different case of that definition and not equal to y_1. (Such a y can always be chosen from a 3-element set such as $\{[\![\mathsf{ERR}]\!], [\![\mathsf{VAR}\ 0]\!], [\![\mathsf{CON}\ c]\!]\}$ for some fixed $c \in L_{con}$.)

Now define a function g_3 to agree with g_1 and g_2 in all components except i and j, and with $\pi_j \circ g_3 = \mathrm{id}$ while $\pi_i \circ g_3$ is a constant function with value y. This

function again satisfies the conditions of Definition 3.54. By construction, $f \circ g_3(y_1) = f \circ g_1(y) = y$ is from a different case of Definition 3.53 than $f \circ g_3(y_2) = f \circ g_2(y)$, so by expand_abstr, $f \circ g_3$ must be the identity function; however, this contradicts $y_1 \neq y$. So our original assumption must be false, and we must have $f = \pi_i$ as claimed.

Since $[\![\sigma\,.\,x_i]\!] = \pi_i$, the result follows in this case; it remains to prove it when $f \circ g$ is *not* the identity function for any such g. We consider cases for $f(y_0, y_0, \ldots, y_0)$ according to expr_nchotomy (Lemma 3.28):

- If $f(y_0, y_0, \ldots, y_0) = [\![\mathsf{CON}\ c]\!]$ for some $c \in L_{con}$, then we claim that f is a constant function. Indeed, suppose not; then it must take values $v_1 = [\![\mathsf{CON}\ c]\!]$ and $v_2 \neq [\![\mathsf{CON}\ c]\!]$ at some points Y_1 and Y_2 in its domain. We may assume, without loss of generality, that Y_1 and Y_2 differ only in the i^{th} component. Define a function $g\colon [\![expr]\!] \to [\![expr]\!]^n$ such that $\pi_i \circ g = \mathrm{id}$ while the other projections are constant functions agreeing with Y_1 and Y_2. By Definition 3.54, we have $[\![\mathsf{abstr}]\!](f \circ g) = 1$, while by assumption, $f \circ g \neq \mathrm{id}$. By expand_abstr (Lemma 3.39) and expr_distinct (Lemma 3.22), the only case that agrees with $f \circ g \circ \pi_i(Y_0) = v_0$ is $f \circ g = [\![\lambda x.\,\mathsf{CON}\ c]\!]$; but this is contradicted by $f \circ g \circ \pi_i(Y_1) = v_1$. Thus, our original assumption must be false and f must be a constant function as claimed. We then have $\big[\!\big[\sigma\,.\,(\mathsf{CON}\ c)\big]\!\big] = f$, and the result follows.

  The VAR and ERR cases are similar.

- If $f(y_0, y_0, \ldots, y_0) = [\![\mathsf{APP}]\!](y_1, y_2)$, then by a similar argument, $f(Y)$ is of the form $[\![\mathsf{APP}]\!](v_1, v_2)$ for all $Y \in [\![expr]\!]^n$. Thus, we may define functions $f_1, f_2\colon [\![expr]\!]^n \to [\![expr]\!]$ such that $f(Y) = [\![\mathsf{APP}]\!](f_1(Y), f_2(Y))$ for all Y. We have $f_1, f_2 \in M^n_{expr}$, taking the necessary abstr conditions from f via abstr_APP (Lemma 3.13). By size_APP (Lemma 3.31), we may apply the induction hypothesis to obtain terms $t_1, t_2 \in L^{\sigma}_{expr}$ such that $[\![\sigma\,.\,t_i]\!] = f_i$ for

$i \in \{1, 2\}$. From the semantics of Isabelle/HOL function application, we have $[\![\sigma\,.\,(\mathsf{APP}\ t_1\ t_2)]\!] = f$, and again the result follows.

- The final case is $f(y_0, y_0, \ldots, y_0) = [\![\mathsf{LAM}]\!](h)$ where $h\colon [\![expr]\!] \to [\![expr]\!]$ satisfies $[\![\mathsf{abstr}]\!](h) = 1$. Again by a similar argument, we obtain a functional H such that for all $Y \in [\![expr]\!]^n$, we have $f(Y) = [\![\mathsf{LAM}]\!](H(Y))$ and $H(Y)$ is a function from $[\![expr]\!]$ to $[\![expr]\!]$ satisfying $[\![\mathsf{abstr}]\!](H(Y)) = 1$. Define a function $H'\colon [\![expr]\!]^{n+1} \to [\![expr]\!]$ by $H'(Y, y) = H(Y)(y)$. We have $H' \in M_{expr}^{n+1}$, with the abstr conditions for the last component given above, and those for the other components taken from f via abstr_LAM (Lemma 3.38). By size_LAM (Lemma 3.32) we may apply the induction hypothesis, using the context $\sigma' = \sigma, x$ for some variable $x :: expr$ not occurring in σ, to obtain a term $t' \in L_{expr}^{\sigma'}$ such that $[\![\sigma'\,.\,t']\!] = H'$. From the semantics of Isabelle/HOL lambda-abstraction, we have $[\![\sigma\,.\,(\lambda x.\, t')]\!] = H$, from which we may deduce $[\![\sigma\,.\,(\mathsf{LAM}\ x.\, t')]\!] = [\![\mathsf{LAM}]\!] \circ H = f$.

In all cases, we have obtained a term $t \in L_{expr}^{\sigma}$ such that $[\![\sigma\,.\,t]\!] = f$, as was to be proven. □

Proof of Theorem 3.56. Well-definedness and bijectivity of the function ϕ_σ given in the statement of the theorem follow directly from Lemmas 3.57, 3.58, 3.59, and 3.60.

Compositionality is just an instance of the term substitution property for $[\![\_]\!]$, as stated in Section 3.4.1 (or [63, Lemma 4]). □

3.4.4 Discussion and Comparison

A representational adequacy result is meant to show correctness of the use of a particular representation of terms for object-language reasoning. This can be broken down into correctness of particular reasoning steps:

- Correctness of equality on Hybrid terms follows from injectivity of ϕ_σ. (For open terms, i.e., when σ is not empty, its variables must be λ-bound.)

- Correctness of quantification over Hybrid terms follows from surjectivity of ϕ_σ. (For open terms, abstr conditions are required as specified in Definition 3.54.[21])

- Correctness of Hybrid's operators on the type *expr* is trivial, once their arguments are translated from M^n_{expr} to L^σ_{expr} using ϕ^{-1}_σ.

- Correctness of substitution by function application follows from compositionality, together with the semantics of function abstraction and application. (It could also be justified syntactically, since Isabelle/HOL equates β-convertible terms.)

Even large and complicated formalizations such as those of Chapters 5 to 9 manipulate Hybrid terms using these operations, so the correctness of such manipulations follows from Theorem 3.56.

We briefly revisit the assumption of a set L_{con} of terms of type *con* for which $[\![\_]\!]$ is bijective onto $[\![con]\!]$. If it fails, we could recover a "relative adequacy" result by working in a *diagram language* consisting of Isabelle/HOL syntax augmented with a constant for each element of $[\![con]\!]_M$, for a particular model M. These constants could then be used as L_{con}, and we would obtain a compositional bijection as in Theorem 3.56, except that the set in bijection with M^n_{expr} would no longer consist entirely of syntax and the structure of sub-"terms" of type *con* would depend on M.

This dependency on M could be eliminated in the case where T categorically axiomatizes (on standard models) some particular mathematical structure (such as the real numbers, as the unique complete ordered field) in the form of a type *con* with suitable operators. Thus we could still obtain a satisfactory adequacy result in

[21] Except where omitting the abstr conditions strengthens the statement, without making it unprovable; for instance, in object-language elimination rules that take advantage of the stronger form of injectivity for LAM proven in Section 3.2.6.

that case. Stating and proving an adequacy result for such cases *in general* would, however, be an additional complication – one that we chose to avoid in Section 3.4.3.

We now turn to a comparison of our adequacy result (Theorem 3.56) with Crole's result [13]. In addition to trivial differences resulting from the changes to Hybrid described in Section 3.2, there are some more substantial differences:

- **Domain of the function:** Crole's function Π_ϵ is defined on α-equivalence classes of untyped λ-terms, while our function ϕ_σ is defined on α-equivalence classes of a subset of Isabelle/HOL terms of type *expr*.

  Our approach leads to a more direct proof, but perhaps a less elegant result, in the sense that Hybrid is intended to represent λ-calculus syntax rather than specifically Isabelle/HOL syntax. However, it would be straightforward to establish a bijection between the two, and thus prove adequacy of Hybrid for a separately-defined λ-calculus as a corollary of Theorem 3.56.

- **Codomain of the function:** Crole's function Π_ϵ takes values in a logical framework intended as a model of Hybrid, but not proven to correspond to the actual formal theory. Our function ϕ_σ, on the other hand, takes values from an arbitrary set-theoretic model of Hybrid as an Isabelle/HOL theory. In this sense, the present work is a more complete proof of Hybrid's adequacy.

  It should be noted that a corollary of Theorem 3.56 together with Crole's result (updated for the present version of Hybrid) is that Crole's logical-framework model of Hybrid is, in fact, bijective with the set-theoretic semantics of Hybrid.

- **Representation of open terms:** Crole's adequacy result represents open terms using VAR for variables free in the original λ-calculus term, but uses free variables of the logical framework for variables bound in the original λ-calculus term (which appear free when translating its subterms). Our result uses a single representation of open terms, as certain functions from $[\![expr]\!]^n$ to $[\![expr]\!]$.

This results in major differences in the structure of the proof of adequacy, and also represents a difference in *what* is being proved adequate for λ-calculus (or similar) syntax. This difference is motivated by our intention to replace the use of VAR to represent open terms (as in Lemma 3.33, expr_VAR_induct) with an alternate representation such as that given in Section 3.3 (*nexp*, as used in Theorem 3.51, whose integration into Hybrid remains as future work).

- **Prerequisites:** Crole's proof is based on de Bruijn syntax, and requires unfolding of most of Hybrid's definitions (as modeled in a logical framework). Our proof is based solely on the properties that the present version of Hybrid formally proves for the type *expr*, making no reference to the type *dB*, the conversion functions dB :: ($expr \Rightarrow dB$) and expr :: ($dB \Rightarrow expr$), or any of their properties.

  Not only does this allow our proof to be much shorter than Crole's, but it also makes Theorem 3.56 a *stronger* result, in the sense that it represents not just correctness but also a form of completeness for Hybrid's set of lemmas concerning the type *expr*.

  This result was enabled by certain of the improvements to Hybrid described in Section 3.2, notably Lemmas 3.38 (abstr_LAM) and 3.39 (expand_abstr). In effect, part of the work of proving adequacy – a meta-theoretical result – is done in the formal setting of Isabelle/HOL. (These lemmas also turned out to be useful for other purposes, as described in Section 3.2.9.)

3.5 Related Work

Hybrid is unique among approaches to formalizing variable-binding constructs in that it is based on full higher-order abstract syntax, and it is built definitionally in a general-purpose proof assistant (Isabelle/HOL).

Thus, there are three categories of related work. One is those approaches that make use of HOAS, but not in a general-purpose proof assistant (or not definitionally). The second is those approaches that formalize variable-binding constructs in a proof assistant, but not using full HOAS. The third is related work on Hybrid itself, which has been developed by various people starting with Ambler, Crole, and Momigliano's work in [2].

Related work for Hybrid is discussed in great detail by Felty and Momigliano in [21]; we consider here only a subset of the many approaches that have been explored.

Other approaches to formalizing HOAS

One approach to formalizing HOAS is the formalization of *logical frameworks*, such as LF [32]. Pfenning and Schürmann's *Twelf* system [60] is one that takes this approach. It implements the specification layer of LF (a form of type theory) with a sophisticated type-inference algorithm that minimizes notational overhead. For meta-reasoning, it uses a form of constraint logic programming on LF terms, together with algorithms for verifying the totality of relations defined as logic programs, so that they may be interpreted as proofs of meta-theoretic properties.

A related approach based on *functional* programming rather than logic programming is Pientka and Dunfield's work [61]. This work has focused on programming with HOAS rather than meta-reasoning, but Twelf's approach to proving meta-theoretic properties by totality checking of programs could be used here as well. (That approach exploits the expressive power of dependent types, so that the type of a program specifies the property to be proved.)

Another approach is based on logics with definitional reflection, starting with McDowell and Miller's work on $FO\lambda^{\Delta\mathbb{N}}$ [46, 47]. Successive enhancements to $FO\lambda^{\Delta\mathbb{N}}$ have led to Linc (Momigliano and Tiu, [53]) and later $\mathcal{G}$ (Gacek et al., [25]), which has been used by Gacek as the basis for an interactive theorem prover called *Abella*

[24]. This system uses the two-level approach that originated with McDowell and is also used with Hybrid. It has the advantage of being purpose-built for reasoning about formal systems, but this can also be a disadvantage in that it cannot exploit the extensive libraries of formalized mathematics available for proof assistants such as Isabelle/HOL and Coq.

Other representations of bound variables in proof assistants

There are projects that aim to implement in Isabelle/HOL some of the other general approaches to representation of variable-binding constructs mentioned in Chapter 1. Most prominent among these is Urban's *nominal datatype* package [73], which seeks to formalize equivalence classes of terms up to renaming of bound variables, and also the Barendregt variable convention, using concepts from Gabbay and Pitts' *nominal logic* [23, 64].

Nominal, unlike the present version of Hybrid, provides *typed* abstract syntax similar to Isabelle's **datatype** package. It also has better-developed induction principles than the present version of Hybrid, although our work in Section 3.3 is a step in that direction. On the other hand, Nominal requires substitution to be defined recursively and its basic properties developed, where in Hybrid substitution can be defined by simple function application (for which many useful properties are already provided by Isabelle/HOL).

Another implementation of variable binding in Isabelle/HOL is Andrew Gordon's work [29]. It builds a representation of named-variable syntax up to renaming of bound variables (i.e., of equivalence classes) definitionally in terms of de Bruijn indices, and it was the starting point for the development of Hybrid [2].

There are also *weak HOAS* approaches that use a function space to represent variable binding, but one in which the argument type is different from the type of terms. Unlike *full HOAS* as in Hybrid, which represents variable binding using

endofunctions of the type of terms, the weak HOAS approach does not violate the constraints for an inductive datatype. It is still necessary to deal with the problem of functions that do not represent syntax, but they can be excluded by other means than predicates, e.g., by the use of polymorphic types in Chlipala's work in Coq [10]. Weak HOAS is related to mathematical models of variable binding based on presheaf categories [22, 34].

Other work on Hybrid

In addition to the Isabelle/HOL theory of [2] that was the starting point for Hybrid as described in this chapter, there are several versions of Hybrid based on the Coq proof assistant. One such version [18, 21] closely follows the structure of the Isabelle/HOL version, and uses non-constructive features via Coq's ClassicalChoice library. There is also a constructive variant of Hybrid for Coq, introduced by Capretta and Felty in [8], and extended in [7] to a *typed* version that can be instantiated with *signatures* for various object languages, specifying multiple sorts and operations.

As well as these variants of Hybrid, there have also been a number of applications and case studies for Hybrid, which will be discussed in Section 10.2. There is also Crole's work on adequacy for Hybrid [13], which was compared with our adequacy result (Theorem 3.56) in Section 3.4.4.

Chapter 4

Mini-ML with References

In preparation for presenting our case study in Hybrid in Chapters 5 to 10, we describe the object language whose formalization is the subject of the case study. It is a variant of Mini-ML [12] with mutable references. This object language is taken from an example in Cervesato and Pfenning's work on a linear logical framework [9], and we closely follow its presentation there. (Definition 4.3 is almost exactly as shown in [9], while Definition 4.2, Table 4.2, and Table 4.1 agree with [9] but fill in many cases omitted there.) See also [62] for general background on typing and operational semantics.

4.1 Syntax

We define abstract syntax for Mini-ML expressions and types using a BNF-like notation. Each syntactic class is denoted by a specific metavariable: v for values, e for expressions, x for variables, c for abstract memory cell locations, and τ for types. Subscripts and primes are used to distinguish separate instances of the same syntactic class. Infix notations are used for some constructs to aid readability. Parentheses are introduced as needed to remove ambiguity.

Definition 4.1 (Values)

$v ::= 0 \mid \mathsf{suc}\ v \mid \mathsf{true} \mid \mathsf{false} \mid * \mid (v_1, v_2) \mid \lambda x.\, e \mid x \mid \mathsf{ref}\ c$

Mini-ML values consist of natural numbers in unary notation (0 and suc), booleans (true and false), unit ($*$), pairs, functions, variables, and memory references. Although pairs are written in the usual mathematical notation as (v_1, v_2), the prefix notation ($\mathsf{pair}\ v_1\ v_2$) is used synonymously where convenient. The body e of a function ($\lambda x.\, e$) can be an arbitrary expression, not necessarily a value. Variables x denote function arguments, and they are considered values because Mini-ML uses call-by-value evaluation; however, they are never actually used in a context that requires a value rather than a general expression. Memory references ($\mathsf{ref}\ c$) arise during evaluation and may not appear in the text of a program. Indeed, no specific syntax is defined for memory locations c; rather, they are treated as *location variables*, to be made local by binding operators, and assumed merely to form a countably infinite set disjoint from *program variables* x.

Definition 4.2 (Expressions)

$$\begin{aligned} e ::= {} & 0 \mid \mathsf{suc}\ e \mid \mathsf{pred}\ e \mid \mathsf{iszero}\ e \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\ & \mid * \mid (e_1, e_2) \mid \pi_1\ e \mid \pi_2\ e \mid e_1\ e_2 \mid \lambda x.\, e \mid x \mid \mathsf{fix}\, x.\, e \\ & \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{letv}\ x = v\ \mathsf{in}\ e \mid \mathsf{let\_rec}\ x = e_1\ \mathsf{in}\ e_2 \\ & \mid \mathsf{ref}\ e \mid !\, e \mid e_1 := e_2 \mid \mathsf{ref}\ c \end{aligned}$$

Expressions extend the syntax of values with some basic operations: predecessor and zero-test functions for natural numbers, a conditional construct for booleans, projections for pairs, application for functions, recursion, definition constructs, and reference manipulation.

The letv construct allows "ML-polymorphism", in that the typing rules will not require v to have the same type in all the places where it is substituted for x in e. (It substitutes only values, not general expressions, to avoid trouble with type safety in the presence of mutable references; see [62, § 22.7].) The let_rec construct

allows recursion, i.e., x is bound in e_1 as well as e_2. It is syntactic sugar for the fixpoint construct fix, which binds a variable x that recursively stands for the entire fix-expression.

There are three basic operations on references: allocation, dereferencing, and assignment. (ref e) allocates a new location c, and evaluates to the value (ref c). $!e$ (also written deref e) expects e to be a reference, and returns the value associated with its location. $e_1 := e_2$ (also written assign e_1 e_2) expects e_1 to be a reference, and stores at its location the value that results from evaluating e_2. There is no deallocation operation; an implementation would presumably use garbage collection. The allocation mechanism is not specified except that it should always return a *fresh* location. The representation of values in memory is likewise considered an implementation detail.

Every value is also an expression, however it would not be enough to have a case $e ::= v$ because constructs such as suc may be applied to expressions as well as values.

4.2 Typing

Definition 4.3 (Types)

$\tau ::= \mathit{nat} \mid \mathit{bool} \mid \mathit{1} \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid \tau\ \mathit{ref}$

Mini-ML is equipped with simple types, consisting of basic types for natural numbers, booleans, and unit, together with type constructors for pairs, functions, and references.

Definition 4.4 (Typing Contexts)

$\Gamma ::= \cdot \mid \Gamma, x{:}\tau \mid \Gamma, c{:}\tau$

The typing judgment $\Gamma \vdash e : \tau$ means that expression e has type τ in the typing context Γ. It is defined recursively by the rules in Table 4.1, which is based on Figure 4 of [9]. Γ gives types for function arguments and reference cells occurring free in e.

Natural numbers and booleans

$$\frac{}{\Gamma \vdash 0 : nat}\ \textbf{ofe\_zero} \qquad \frac{\Gamma \vdash e : nat}{\Gamma \vdash \mathsf{suc}\ e : nat}\ \textbf{ofe\_suc}$$

$$\frac{\Gamma \vdash e : nat}{\Gamma \vdash \mathsf{pred}\ e : nat}\ \textbf{ofe\_pred} \qquad \frac{\Gamma \vdash e : nat}{\Gamma \vdash \mathsf{iszero}\ e : bool}\ \textbf{ofe\_iszero}$$

$$\frac{}{\Gamma \vdash \mathsf{true} : bool}\ \textbf{ofe\_true} \qquad \frac{}{\Gamma \vdash \mathsf{false} : bool}\ \textbf{ofe\_false}$$

$$\frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau}\ \textbf{ofe\_ifthen}$$

Unit and pairs

$$\frac{}{\Gamma \vdash * : 1}\ \textbf{ofe\_unit}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}\ \textbf{ofe\_pair} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i\ e : \tau_i}\ \textbf{ofe\_proj}$$

Functions and recursion

$$\frac{\Gamma, x{:}\tau' \vdash e : \tau}{\Gamma \vdash \lambda x.\, e : \tau' \to \tau}\ \textbf{ofe\_lam} \qquad \frac{x \notin \Gamma'}{\Gamma, x{:}\tau, \Gamma' \vdash x : \tau}\ \textbf{ofe\_id}$$

$$\frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}\ \textbf{ofe\_app} \qquad \frac{\Gamma, x{:}\tau \vdash e : \tau}{\Gamma \vdash \mathsf{fix}\, x.\, e : \tau}\ \textbf{ofe\_fix}$$

Definitions

$$\frac{\Gamma \vdash [v/x]e : \tau}{\Gamma \vdash \mathsf{letv}\ x = v\ \mathsf{in}\ e : \tau}\ \textbf{ofe\_letv} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}\ \textbf{ofe\_let}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let\_rec}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}\ \textbf{ofe\_letrec}$$

References

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{ref}\ e : \tau\ ref}\ \textbf{ofe\_ref} \qquad \frac{\Gamma \vdash e : \tau\ ref}{\Gamma \vdash\, !\, e : \tau}\ \textbf{ofe\_deref}$$

$$\frac{\Gamma \vdash e_1 : \tau\ ref \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : 1}\ \textbf{ofe\_assign} \qquad \frac{}{\Gamma, c{:}\tau, \Gamma' \vdash \mathsf{ref}\ c : \tau\ ref}\ \textbf{ofe\_cell}$$

Table 4.1: Typing rules for Mini-ML

The rules are mostly self-explanatory. However, it should be noted that without type annotations on bound variables, the type τ is not uniquely determined by Γ and e. For example, the expression $(\lambda x.\, x)$ can be typed as either $\mathsf{bool} \to \mathsf{bool}$ or $\mathsf{nat} \to \mathsf{nat}$ (among others) in the empty context.

Implementations of ML-like languages typically use type variables, so that each expression can be assigned a unique *principal type* from which all its valid types can be obtained as instances. (See [62, § 22.5].) However, such complications are not needed for the present purposes.

Also, the rule **ofe_letv** might never type-check the substituted value v if the variable x does not occur in e. Practical programming languages usually do require it to be well-typed even in this case.

The condition on the rule **ofe_id** ensures that scopes of variables are respected. It is not necessary at this point to deal with α-conversion in a more systematic way.

Further typing rules will be needed to type-check intermediate states of a computation; these will be given after the formal sematics, in Section 4.4.

4.3 Continuation-Style Semantics

The operational semantics of Mini-ML are formalized as a continuation-style big-step evaluation judgment. We begin by defining additional syntactic classes to represent intermediate steps of a computation.

Definition 4.5 (States)

$S ::= \cdot \mid S, c = v$

A state is an association list specifying values for reference cells. The cells in the list are required to be distinct.

Definition 4.6 (Continuations)

$K ::= \mathsf{init} \mid K, \lambda x.\, e$

A continuation is a list of function expressions. It serves as a stack of functions and primitives that are waiting for their arguments to be evaluated.

Definition 4.7 (Answers)

$w ::= (S, v) \mid \mathsf{new}\ c.\, w$

An answer consists of a final state and a value, in which location variables may be bound by new. This allows the result value to refer to reference cells allocated in the course of evaluation, without giving those cells global names.

The evaluation judgment then has the form $S \triangleright K \vdash e \hookrightarrow w$, where S is the initial state, K is the continuation, e is the expression to be evaluated, and w is the result of evaluation. It is defined by the rules in Table 4.2, which is based on Figure 5 of [9].

Values are handled by **ex_return**, which takes the last function from a continuation, applies it to the value, and recursively evaluates the result; or if the continuation is empty, by **ex_init**, which packages the current state and the value as the final result.

Evaluation of arguments for functions and primitives is performed by rules like **ex_suc** and **ex_pred**. The argument to be evaluated is taken out of its context and becomes the new expression to evaluate, while the context is turned into a function expression and appended to the continuation.

There are two cases where evaluation is deferred. One is in the body of a function: a function $(\lambda x.\, e)$ is considered a value even if its body e is not. The other is in the conditional construct (if), which evaluates its boolean argument but then chooses between its then and else clauses before proceeding with evaluation. Such deferred evaluation is necessary to allow programs to control side effects, and even to have terminating recursive functions.

The remaining rules specify how operations are actually performed once their arguments have been evaluated. For instance, **ex_pred0** and **ex_predS** define the

Continuations

$$\frac{S \triangleright K \vdash [v/x]e \hookrightarrow w}{S \triangleright K; \lambda x.\, e \vdash v \hookrightarrow w}\ \textbf{ex\_return} \qquad \frac{}{S \triangleright \mathsf{init} \vdash v \hookrightarrow (S, v)}\ \textbf{ex\_init}$$

Natural numbers and booleans

$$\frac{S \triangleright K; \lambda x.\, \mathsf{suc}\ x \vdash \bar{e} \hookrightarrow w}{S \triangleright K \vdash \mathsf{suc}\ \bar{e} \hookrightarrow w}\ \textbf{ex\_suc} \qquad \frac{S \triangleright K; \lambda x.\, \mathsf{pred}\ x \vdash \bar{e} \hookrightarrow w}{S \triangleright K \vdash \mathsf{pred}\ \bar{e} \hookrightarrow w}\ \textbf{ex\_pred}$$

$$\frac{S \triangleright K \vdash 0 \hookrightarrow w}{S \triangleright K \vdash \mathsf{pred}\ 0 \hookrightarrow w}\ \textbf{ex\_pred0} \qquad \frac{S \triangleright K \vdash v \hookrightarrow w}{S \triangleright K \vdash \mathsf{pred}\ (\mathsf{suc}\ v) \hookrightarrow w}\ \textbf{ex\_predS}$$

$$\frac{S \triangleright K; \lambda x.\, \mathsf{iszero}\ x \vdash \bar{e} \hookrightarrow w}{S \triangleright K \vdash \mathsf{iszero}\ \bar{e} \hookrightarrow w}\ \textbf{ex\_iszero} \qquad \frac{S \triangleright K; \lambda x.\, \mathsf{if}\ x\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \vdash \bar{e}_1 \hookrightarrow w}{S \triangleright K \vdash \mathsf{if}\ \bar{e}_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \hookrightarrow w}\ \textbf{ex\_ifthen}$$

$$\frac{S \triangleright K \vdash \mathsf{true} \hookrightarrow w}{S \triangleright K \vdash \mathsf{iszero}\ 0 \hookrightarrow w}\ \textbf{ex\_iszero0} \qquad \frac{S \triangleright K \vdash e_1 \hookrightarrow w}{S \triangleright K \vdash \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \hookrightarrow w}\ \textbf{ex\_ifthenT}$$

$$\frac{S \triangleright K \vdash \mathsf{false} \hookrightarrow w}{S \triangleright K \vdash \mathsf{iszero}\ (\mathsf{suc}\ v) \hookrightarrow w}\ \textbf{ex\_iszeroS} \qquad \frac{S \triangleright K \vdash e_2 \hookrightarrow w}{S \triangleright K \vdash \mathsf{if}\ \mathsf{false}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \hookrightarrow w}\ \textbf{ex\_ifthenF}$$

Pairs

$$\frac{S \triangleright K; \lambda x.\, (x, e_2) \vdash \bar{e}_1 \hookrightarrow w}{S \triangleright K \vdash (\bar{e}_1, e_2) \hookrightarrow w}\ \textbf{ex\_pair} \qquad \frac{S \triangleright K; \lambda x.\, (v, x) \vdash \bar{e} \hookrightarrow w}{S \triangleright K \vdash (v, \bar{e}) \hookrightarrow w}\ \textbf{ex\_pair1}$$

$$\frac{S \triangleright K; \lambda x.\, \pi_i\ x \vdash \bar{e} \hookrightarrow w}{S \triangleright K \vdash \pi_i\ \bar{e} \hookrightarrow w}\ \textbf{ex\_proj} \qquad \frac{S \triangleright K \vdash v_i \hookrightarrow w}{S \triangleright K \vdash \pi_i\ (v_1, v_2) \hookrightarrow w}\ \textbf{ex\_proj1}$$

Functions and recursion

$$\frac{S \triangleright K; \lambda x.\ x\ e_2 \vdash \bar{e}_1 \hookrightarrow w}{S \triangleright K \vdash \bar{e}_1\ e_2 \hookrightarrow w}\ \textbf{ex\_app} \qquad \frac{S \triangleright K; \lambda x.\ v\ x \vdash \bar{e} \hookrightarrow w}{S \triangleright K \vdash v\ \bar{e} \hookrightarrow w}\ \textbf{ex\_app1}$$

$$\frac{S \triangleright K \vdash [v/x]e \hookrightarrow w}{S \triangleright K \vdash (\lambda x.\, e)\, v \hookrightarrow w}\ \textbf{ex\_app2} \qquad \frac{S \triangleright K \vdash [(\mathsf{fix}\, x.\, e)/x]e \hookrightarrow w}{S \triangleright K \vdash \mathsf{fix}\, x.\, e \hookrightarrow w}\ \textbf{ex\_fix}$$

Definitions

$$\frac{S \triangleright K \vdash (\lambda x.\, e_2)e_1 \hookrightarrow w}{S \triangleright K \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \hookrightarrow w}\ \textbf{ex\_let} \qquad \frac{S \triangleright K \vdash [v/x]e \hookrightarrow w}{S \triangleright K \vdash \mathsf{letv}\ x = v\ \mathsf{in}\ e \hookrightarrow w}\ \textbf{ex\_letv}$$

$$\frac{S \triangleright K \vdash (\lambda x.\, e_2)(\mathsf{fix}\, x.\, e_1) \hookrightarrow w}{S \triangleright K \vdash \mathsf{let\_rec}\ x = e_1\ \mathsf{in}\ e_2 \hookrightarrow w}\ \textbf{ex\_let\_rec}$$

Table 4.2: Evaluation rules for Mini-ML

References

$$\frac{S \triangleright K; \lambda x.\,\mathsf{ref}\ x \vdash \bar{e} \hookrightarrow w}{S \triangleright K \vdash \mathsf{ref}\ \bar{e} \hookrightarrow w}\ \textbf{ex\_ref} \qquad \frac{S, c = v \triangleright K \vdash \mathsf{ref}\ c \hookrightarrow w}{S \triangleright K \vdash \mathsf{ref}\ v \hookrightarrow \mathsf{new}\ c.\,w}\ \textbf{ex\_ref1}$$

$$\frac{S \triangleright K; \lambda x.\,!\,x \vdash \bar{e} \hookrightarrow w}{S \triangleright K \vdash !\,\bar{e} \hookrightarrow w}\ \textbf{ex\_deref} \qquad \frac{S, c = v, S' \triangleright K \vdash v \hookrightarrow w}{S, c = v, S' \triangleright K \vdash !\,(\mathsf{ref}\ c) \hookrightarrow w}\ \textbf{ex\_deref1}$$

$$\frac{S \triangleright K; \lambda x.\,x := e_2 \vdash \bar{e}_1 \hookrightarrow w}{S \triangleright K \vdash \bar{e}_1 := e_2 \hookrightarrow w}\ \textbf{ex\_assign} \qquad \frac{S \triangleright K; \lambda x.\,v := x \vdash \bar{e} \hookrightarrow w}{S \triangleright K \vdash v := \bar{e} \hookrightarrow w}\ \textbf{ex\_assign1}$$

$$\frac{S, c = v, S' \triangleright K \vdash * \hookrightarrow w}{S, c = v_0, S' \triangleright K \vdash \mathsf{ref}\ c := v \hookrightarrow w}\ \textbf{ex\_assign2}$$

Table 4.2: Evaluation rules for Mini-ML, continued

predecessor function for natural numbers by cases on its argument. (There are no such rules for suc because $\mathsf{suc}\ v$ is already a value when v is.) The rule **ex_app2** is the usual β-reduction for functions, and the notation $[v/x]e$ represents capture-avoiding substitution (renaming bound variables as needed) of v for x in e.

The ref operation is performed by the rule **ex_ref1**, which introduces a new location variable c to represent the newly allocated reference cell. This location must be *fresh*; the requirement that cells in a state be distinct ensures that it does not occur in S. It is bound by new in the answer, so that free location variables are not introduced by evaluation. It is returned to the continuation as a value of the form $\mathsf{ref}\ c$, a form that is not used directly in programs.

Dereferencing (!) is performed by the rule **ex_deref1**, by looking up the location to be dereferenced in the state, and returning the corresponding value. Assignment (:=) is performed by the rule **ex_assign2**, by looking up the location to be assigned and replacing its value in the state with the right-hand side of the assignment.

This continuation-based approach, with its rules for evaluating arguments, makes the call-by-value evaluation order explicit. Indeed, evaluation is deterministic except for the possibility of re-evaluating arguments that are already values. (That possi-

bility can be eliminated, without changing the set of derivable typing statements, by requiring the expressions written as $\bar{e}$ in Table 4.2 to *not* be values.)

4.4 Type Safety

A key property of a typed language is *type safety*, which can be informally stated as "Well-typed programs cannot go wrong". More specifically, such programs should not attempt nonsensical operations, like $(\pi_1\ (\mathsf{suc}\ 0))$, for which there is no evaluation rule; and the result of evaluation should have the same type as the original expression. These properties are called *progress* and *preservation*, respectively.

The type safety properties are most easily stated for a small-step operational semantics, which uses a formal judgment to represent an individual step of evaluation. A big-step operational semantics like that of Section 4.3 formalizes only the complete evaluation of an expression to a value, so properties of the evaluation judgment say nothing directly about the intermediate states of a computation. Structurally inductive proofs of such properties, however, do consider the intermediate states and typically correspond closely with small-step versions. While this may not be entirely satisfying from a mathematical point of view, it works well enough in practice.

A bigger problem, however, is that a straightforward statement of progress for big-step evaluation is *false* for languages with non-terminating computations, such as Mini-ML with let_rec. The evaluation judgment does not distinguish between failure to progress (due to an ill-typed operation) and infinite recursion. (See [62], p. 505.) There are several ways to deal with this problem:

- Add explicit error terms, and rules evaluating ill-typed operations to them.
- State progress as a property of evaluation derivations, rather than just their conclusions.
- Switch to a small-step semantics.

Continuations

$$\frac{}{\Gamma \vdash \mathsf{init} : \tau \to \tau}\ \textbf{ofk\_init} \qquad \frac{\Gamma \vdash e : \tau_1 \to \tau_3 \quad \Gamma \vdash K : \tau_3 \to \tau_2}{\Gamma \vdash K; e : \tau_1 \to \tau_2}\ \textbf{ofk\_func}$$

States

$$\frac{}{\Delta \vdash \cdot : \cdot}\ \textbf{ofc\_empty} \qquad \frac{\Delta \vdash S : \Delta' \quad \Delta \vdash v : \tau}{\Delta \vdash (S, c = v) : (\Delta', c{:}\tau)}\ \textbf{ofc\_cell}$$

Answers

$$\frac{\Delta \vdash S : \Delta \quad \Delta \vdash v : \tau}{\Delta \vdash (S, v) : \tau}\ \textbf{off\_loc} \qquad \frac{\Delta, c{:}\tau' \vdash w : \tau}{\Delta \vdash \mathsf{new}\ c.\, w : \tau}\ \textbf{off\_new}$$

Table 4.3: Typing rules for Mini-ML, continued

- Simply do without a progress property.

For this case study, the latter approach was taken: only a preservation property, also known as *subject reduction*, was considered. To state this property, it is necessary to type states, continuations, and answers as well as expressions. These additional typing judgments are defined in Table 4.3. They use a restricted kind of typing context called a *store typing*, and written Δ, which gives values only for reference cells and not for program variables:

Definition 4.8 (Store Typings)

$\Delta ::= \cdot \mid \Delta, c{:}\tau$

The typing judgment for continuations has the form $\Gamma \vdash K : \tau_1 \to \tau_2$, which means that the continuation K expects a value of type τ_1 and produces an answer of type τ_2. We view τ_1 and τ_2 as separate arguments of this judgment, with the arrow between them being part of its syntax rather than a function type constructor, although semantically it is analogous to one.

The typing judgment for states has the form $\Delta \vdash S : \Delta'$. It means that the same locations occur in S and Δ', and if a location c is associated with a value v in S and

with a type t in Δ', then v is well-typed with type t in the context Δ.

We will typically be interested in typing statements of the form $\Delta \vdash S : \Delta$, with the same store typing on both sides. However, in deriving such a statement, it is necessary to check the reference cells one-by-one, using an axiom (**ofc_empty**) to start with the empty state and a rule (**ofc_cell**) to check each cell as it is added to the state. To ensure that the locations in Δ correspond one-to-one with those in S, the state and store typing should be built up simultaneously. Yet we need the entire store typing, in general, to check even the first cell, since that cell may have been assigned a value containing references to newer cells. These opposing requirements are the reason for having two separate store-typing arguments, which are treated differently by the rule **ofe_cell**.

The typing judgment for answers has the form $\Delta \vdash w : \tau$. The type of an answer is the type of the value it contains, but the judgment also checks the types of values in reference cells, allowing those bound by new to have any type so long as it is used consistently.

Now we have everything required to state a subject reduction property.

Theorem 4.9 (Subject Reduction) *Suppose $S \triangleright K \vdash e \hookrightarrow w$. If the state S is well-typed, $\Delta \vdash S : \Delta$, under a store typing Δ; the continuation K is well-typed, $\Delta \vdash K : \tau \to \tau'$; and the expression e is well-typed, $\Delta \vdash e : \tau$; then the answer w is also well-typed: $\Delta \vdash w : \tau'$.*

The proof is by induction on the evaluation judgment, and is mostly straightforward. The structure of the first formalized version of subject reduction (in Chapter 5) is similar to what would be used in an informal proof.

Chapter 5

Case Study: One-level Approach

In this and the following chapters, we present a case study consisting of five different formalizations of subject reduction for Mini-ML with references, as described in Chapter 4, using the proof assistant Isabelle/HOL and the Hybrid package as presented in Chapter 3. For the purpose of this presentation, the term *object logic* (OL) will refer to Mini-ML with references.

The purpose of the case study was to explore the issues that arise with different specification logics and encoding styles. Although the size and composition of the formal proofs are compared in Chapter 10, it should be kept in mind that the formalizations were not done independently – rather, each was built as a modification of the previous one. Additionally, some simplifications and adjustments for consistency were made *after* all of the formalizations were complete.

As in the presentation of Hybrid in Chapter 3, we omit most of the proofs and some technical lemmas. The later chapters will refer to the earlier ones for the parts of the formalization that were reused. Also, the earlier formalizations are presented in somewhat greater detail than the later ones.

All five formalizations represented OL terms as Hybrid terms of type *expr*. Hybrid was not directly involved in the encoding of judgments, however some of

the arguments of those judgments were Hybrid terms, and some of the formalizations used specification logics supporting quantification over Hybrid terms.

The first version of the formalization used a one-level approach, where OL judgments are represented directly by Isabelle/HOL predicates, with explicit contexts where necessary. This was reasonably straightforward, and provides a baseline against which the later versions can be compared.

All the remaining versions used the two-level approach [19, 21, 20, 49], in which a *specification logic* (SL) is encoded in Isabelle/HOL and OL judgments are represented as SL atoms. They differ in two respects: the particular SL used, and which judgments are actually encoded in the SL (with the others remaining as Isabelle/HOL predicates).

Three different specification logics were used. The first is a minimal version of intuitionistic logic with a backchaining rule [46, § 5.1], previously used in Hybrid in [49, § 4.1] and [21, § 4.1]. It was used for the formalization in Chapter 6, representing typing for functions using a hypothetical judgment. Evaluation was represented at the meta-level as in Chapter 5.

The second specification logic [46, § 5.2] combines intuitionistic and linear logic by the use of two separate contexts. It was used in two formalizations, in Chapters 7 and 8. The first one represents typing for reference cells in the SL context, while reusing the representation of evaluation at the meta-level. The second also represents evaluation as an SL predicate, with the *values* of reference cells in the SL context. Although these formalizations use the same specification *logic*, its *formalization* is different in Chapter 8 to support induction on the height of SL derivations.

The third specification logic adds a third context with *ordered logic* features [65, 66]. It is similar to the SL used in [21, § 5.1], except that the latter omits the linear context. It was used for the formalization in Chapter 9, which encodes continuations in the SL's ordered context as in [21, § 5.3], and otherwise follows the same approach as Chapter 8.

In this and the following chapters, the term "formal" will be reserved for machine-checked proof, i.e., the content of Isabelle/HOL theory files. Ordinary mathematical specifications such as those of Chapter 4 may be called "informal" by contrast, even when they might normally be considered "formal systems".

Every definition, lemma, and theorem stated in these chapters is formalized in the corresponding Isabelle/HOL theory file, and these files are available online [45].

5.1 Syntax

The one-level formalization described here is quite close to the informal presentation in Chapter 4, so this chapter is heavy on formal proof text, many of the general issues having already been addressed.

We used as a starting point an Isabelle/HOL theory called `mlLang.thy` [67], provided by Alberto Momigliano in a personal communication, which formalized subject reduction for a fragment of Mini-ML using an intuitionistic specification logic. We follow the general approach to representing syntax used in previous work on Hybrid.

Most of the Mini-ML syntactic classes will be represented using Hybrid terms. Each syntactic class will be given its own type name, and the use of these type names will specify the *intended* syntactic classes of variables, function and predicate arguments, etc. But Hybrid is untyped, so these type names will actually be abbreviations for a single underlying type, *con expr*.

Functions and predicates will not, in general, constrain their arguments to the appropriate syntactic classes. The statements of lemmas and theorems will be responsible for constraining Hybrid variables to the appropriate syntactic classes where necessary. However, the straightforward definitions of predicates tend to provide some constraints, which will be exploited to prove lemmas and theorems without explicit syntactic constraints – indeed, without ever *defining* formal syntactic validity

predicates (but see the discussion following Definitions 5.2 and 5.20). As long as all variables are universally quantified, it would be trivial to prove properly-constrained corollaries, given the predicates needed to state them.

The type *con* is set up with the constants needed for all of the syntactic classes:

Definition 5.1 (Constants for Hybrid)

datatype *con* = c_eZero | c_eSuc | c_ePred | ... | c_fNew

There is a constant for each construct in Definitions 5.2 and 5.5–5.7, named by prefixing the name of the construct with c_. The names of the constructs themselves start with a lowercase letter indicating the syntactic class. (The letter f is used for answers, rather than w as in Chapter 4.)

Definition 5.2 (Expressions)

types

exp = *con expr*

constdefs

eZero, eTrue, eFalse, eUnit :: *exp*

eZero ≡ CON c_eZero (*others similar*)

eSuc, ePred, eIsZero, eFst, eSnd, eRef, eDeref :: *exp* ⇒ *exp*

eSuc n ≡ CON c_eSuc $$ n (*others similar*)

eApp, ePair, eAssign :: [*exp*, *exp*] ⇒ *exp*

eApp f x ≡ CON c_eApp $$ f $$ x (*notation* f $ x)

ePair x y ≡ CON c_ePair $$ x $$ y (eAssign *similar*)

eIfThen :: [*exp*, *exp*, *exp*] ⇒ *exp*

eIfThen c tt ff ≡ CON c_eIfThen $$ c $$ tt $$ ff

eFn, eFix :: (*exp* ⇒ *exp*) ⇒ *exp*

eFn F ≡ CON c_eFn $$ LAM F (*notation* fn x. F x)

eFix F ≡ CON c_eFix $$ LAM F (*notation* fix f. F f)

eLetv :: [*exp*, *exp* ⇒ *exp*] ⇒ *exp*

eLetv v F ≡ CON c_eLetv $$ v $$ LAM F

eCell :: *cell* ⇒ *exp*

eCell c ≡ CON c_eCell $$ c

Each construct of Mini-ML is represented by its corresponding Hybrid constant, joined by Hybrid's $$ operator with the representations of its immediate subterms. Variable binding (e.g., for eFn) is done using HOAS in the form of Hybrid's LAM. Values do not have their own representation, instead being treated as a special case of expressions. However, the case ref c from Definition 4.1 is renamed eCell. (Isabelle/HOL supports overloading, but it could not be used here, because the argument types *cell* and *exp* are really the same type.)

Isabelle's mixfix and binder annotations are used to provide a more familiar syntax for functions: infix $ for application (eApp), and an fn binder for abstraction (eFn). However, because some of the automated proof methods are sensitive to η-expansion, eFn is often used in preference to the fn notation in the formal proofs.

The use of different but synonymous type names for constructor arguments has no formal significance in Isabelle/HOL, but serves to informally specify the intended syntactic class of each argument, and thus which Hybrid terms are valid representations of Mini-ML expressions. (Indeed, eZero etc. are constructors only in this informal sense, as they are not exhaustive for the type $exp = con\ expr$.)

Definition 5.3

syntax

eLet :: $\left[exp,\ exp \Rightarrow exp \right] \Rightarrow exp$

eLetrec :: $\left[exp \Rightarrow exp,\ exp \Rightarrow exp \right] \Rightarrow exp$

translations

eLet x F $\equiv$ (eFn F) $ x

eLetrec E F $\equiv$ (eFn F) $ (eFix E)

As a simplification, the "let" and "let rec" constructs are represented as derived constructs, using Isabelle/HOL abbreviations, rather than as constructors for *exp*. They are intended to be equivalent to the corresponding constructs defined in Chapter 4, although this was not proved. This approach avoids the need for separate cases in the encoding of evaluation and in the subsequent proofs.

Definition 5.4 (Locations)

types

cell = *con expr*

No constructors are defined for *cell*, nor will the predicates encoding evaluation, etc. impose any constraints on what Hybrid expressions can be used as locations. For the formalization it is enough that a countable infinity of locations are available, since Mini-ML's operations on references do not reveal their representation.

Definition 5.5 (Continuations)

types

cont = *con expr*

constdefs

kCons, kApp :: $[\, exp \Rightarrow exp,\ cont \,] \Rightarrow cont$

kCons E k ≡ CON c_kCons $$ LAM E $$ k (kApp *similar*)

kArg, kPair1, kPair2, kAssign1 :: $[\, exp,\ cont \,] \Rightarrow cont$

kArg e k ≡ CON c_kArg $$ e $$ k (*others similar*)

kPred, kIsZero, kFst, kSnd, kRef, kDeref :: $cont \Rightarrow cont$

kPred k ≡ CON c_kPred $$ k (*others similar*)

kIfThen :: $[\, exp,\ exp,\ cont \,] \Rightarrow cont$

kIfThen tt ff k ≡ CON c_kIfThen $$ tt $$ ff $$ k

kAssign2 :: $[\, cell,\ cont \,] \Rightarrow cont$

kAssign2 c k ≡ CON c_kAssign2 $$ c $$ k

kDone :: *cont*

kDone ≡ CON c_kDone

As in Chapter 4, a continuation is a stack that holds operations whose execution has been deferred while their arguments are being evaluated. However, it is not represented as a list of function expressions. Instead there is a specific constructor for each evaluation rule that adds an operation to the continuation. (These will be explained further when the evaluation judgment is formalized.)

Definition 5.6 (States)

types
state = *con expr*
constdefs
sNil :: *state*
sNil ≡ CON c_sNil
sCons :: [*cell*, *exp*, *state*] ⇒ *state*
sCons c v s ≡ CON c_sCons $$ c $$ v $$ s

A state is a mapping from locations to values, represented as an association list built out of Hybrid terms. The use of Hybrid terms, rather than a list or function type (which would otherwise be more convenient), allows locations to be bound by fNew as defined below.

Definition 5.7 (Answers)

types
final = *con expr*
constdefs
fVal :: [*state*, *exp*] ⇒ *final*
fVal s v ≡ CON c_fVal $$ s $$ v
fNew :: [*cell* ⇒ *final*] ⇒ *final*
fNew F ≡ CON c_fNew $$ LAM F

The representation of answers again uses HOAS for variable binding, this time for location variables. Note that location variables can be distinguished from program variables by the contexts where they occur.

Since multiple Hybrid constructors are needed to represent a single Mini-ML construct, induction over Hybrid terms will visit subterms that do not represent Mini-ML terms. This is awkward to deal with in proofs. Fortunately, induction over terms is not required in the proof of subject reduction. As future work, the consistent use of CON, $$, and LAM should make it possible to derive an alternate induction rule that visits only the intended subterms, without specifically tailoring it for Mini-ML.

The types of Mini-ML do not use bound variables, so unlike the other syntactic classes which share the type of Hybrid expressions, they are represented as an Isabelle/HOL datatype.

Definition 5.8 (Types)

datatype tp = tNat | tBool | tUnit | tPair tp tp | tFun tp tp | tRef tp

This definition corresponds case-by-case to Definition 4.3, with infix notations replaced by named (prefix) constructors.

Typing contexts (Γ, Definition 4.4) are represented in two parts, C :: ($exp \times tp$) set and D :: ($cell \times tp$) set, the latter also being used alone to represent store typings (Δ, Definition 4.8). C and D are intended to assign types to program variables and location variables respectively. But since these variables will be represented in HOAS style as universally quantified terms of types exp and $cell$, the Isabelle/HOL types of C and D must allow them to specify Mini-ML types for arbitrary expressions and locations. (Indeed, for arbitrary Hybrid terms, since Hybrid is untyped.)

The use of sets eliminates irrelevant distinctions of order and multiplicity, and allows the use of Isabelle/HOL's many built-in lemmas and tools for set reasoning.

5.2 Auxiliary predicates

Definition 5.9 (s_fresh)

inductive s_fresh :: $\lbrack cell, state \rbrack \Rightarrow bool$
where s_fresh_Nil_I: s_fresh c′ sNil
| s_fresh_Cons_I: $\llbracket$ c′ $\neq$ c; s_fresh c′ s $\rrbracket \Longrightarrow$ s_fresh c′ (sCons c v s)

(s_fresh c s) is true when the cell c does not already occur in the state s. It is used to choose a new cell when evaluating eRef.

Definition 5.10 (s_func)

inductive s_func :: $state \Rightarrow bool$
where s_func_Nil_I: s_func sNil

| s_func_Cons_I: $[\![$ s_fresh c s; s_func s $]\!] \Longrightarrow$ s_func (sCons c v s)

(s_func s) is true when the state s contains at most one occurrence of any given cell. This should be true of any valid state.

Definition 5.11 (s_lookup)

inductive s_lookup :: $[\, state,\ cell,\ exp \,] \Rightarrow bool$
where s_lookup_Cons_I1: s_lookup (sCons c v s) c v
| s_lookup_Cons_I2: s_lookup s c v $\Longrightarrow$ s_lookup (sCons c′ v′ s) c v

(s_lookup s c v) is true when the state s contains the pair (c, v). It is used in evaluating eDeref.

Definition 5.12 (s_assign)

inductive s_assign :: $[\, state,\ cell,\ exp,\ state \,] \Rightarrow bool$
where s_assign_Cons_I1: s_assign (sCons c v s) c v′ (sCons c v′ s)
| s_assign_Cons_I2: s_assign s c v s′ $\Longrightarrow$ s_assign (sCons c′ v′ s) c v (sCons c′ v′ s′)

(s_assign s c v′ s′) is true when the state s contains a pair (c, v) for some v :: *exp*, and s′ is formed by replacing v with v′ in that pair. It is used in evaluating eAssign.

Definition 5.13 (D_fresh)

constdefs D_fresh :: $[\, cell,\ (cell \times tp)\ set \,] \Rightarrow bool$
D_fresh c D $\equiv \forall$ t. (c, t) $\notin$ D

(D_fresh c D) is true when the location c does not occur in the store typing D.

Definition 5.14 (D_func)

constdefs D_func :: $(cell \times tp)\ set \Rightarrow bool$
D_func D $\equiv \forall$ c t t′. (c, t) $\in$ D $\wedge$ (c, t′) $\in$ D $\longrightarrow$ t = t′

(D_func D) is true when the store typing D associates at most one type with each location. This should be true of any valid store typing.

Several basic properties are proved, which are sufficient to manipulate states and store typings for the purpose of subject reduction. (It might be easier if states were

represented using Isabelle/HOL's *list* type, or a function type, but that would not allow locations to be HOAS bound variables.)

Lemma 5.15 (Properties of s_fresh)

s_fresh_lookup_E:
$$[\![\mathsf{s\_fresh}\ c\ s;\ \neg\ P \Longrightarrow \mathsf{s\_lookup}\ s\ c\ v]\!] \Longrightarrow P$$
s_lookup_Cons_func_E:
$$[\![\mathsf{s\_lookup}\ (\mathsf{sCons}\ c'\ v'\ s)\ c\ v;\ \neg\ P \Longrightarrow \mathsf{s\_func}\ (\mathsf{sCons}\ c'\ v'\ s);\ [\![c = c';\ v = v']\!] \Longrightarrow P;\ [\![c \neq c';\ \mathsf{s\_lookup}\ s\ c\ v;\ \mathsf{s\_func}\ s]\!] \Longrightarrow P]\!] \Longrightarrow P$$

The lemma s_fresh_lookup_E allows the elimination of cases with the contradictory assumptions that a location is fresh for a state, and that it is found in that state. The lemma s_lookup_Cons_func_E is a stronger elimination rule for s_lookup in the case of valid states (satisfying s_func).

Lemma 5.16 (Properties of s_assign)

s_fresh_assign_E:
$$[\![\mathsf{s\_fresh}\ c\ s;\ \neg\ P \Longrightarrow \mathsf{s\_assign}\ s\ c\ v\ s']\!] \Longrightarrow P$$
s_assign_Cons_func_E:
$$[\![\mathsf{s\_assign}\ (\mathsf{sCons}\ c'\ v'\ s)\ c\ v\ s';\ \neg\ P \Longrightarrow \mathsf{s\_func}\ (\mathsf{sCons}\ c'\ v'\ s);\ [\![c = c';\ s' = \mathsf{sCons}\ c\ v\ s]\!] \Longrightarrow P;\ \bigwedge s''.\ [\![c \neq c';\ s' = \mathsf{sCons}\ c'\ v'\ s'';\ \mathsf{s\_assign}\ s\ c\ v\ s'';\ \mathsf{s\_func}\ s]\!] \Longrightarrow P]\!] \Longrightarrow P$$
s_assign_fresh:
$$\mathsf{s\_assign}\ s\ c\ v\ s' \Longrightarrow \mathsf{s\_fresh}\ c'\ s' = \mathsf{s\_fresh}\ c'\ s$$
s_assign_func:
$$\mathsf{s\_assign}\ s\ c\ v\ s' \Longrightarrow \mathsf{s\_func}\ s' = \mathsf{s\_func}\ s$$

The lemmas s_fresh_assign_E and s_assign_Cons_func_E are analogous to the lemmas for s_lookup above, while s_assign_fresh and s_assign_func state that the predicates s_fresh and s_func are invariant under assignment of new values to reference cells.

Lemma 5.17 (Properties of D_fresh)

D_fresh_I:
$\forall$ t. (c, t) $\notin$ D $\Longrightarrow$ D_fresh c D
D_fresh_empty_I:
D_fresh c $\varnothing$
D_fresh_insert_I:
$[\![$ c $\neq$ c'; D_fresh c D $]\!] \Longrightarrow$ D_fresh c $\big($D $\cup$ $\{$(c', t)$\}\big)$
D_fresh_E:
$[\![$ D_fresh c D; $\forall$ t. (c, t) $\notin$ D $\Longrightarrow$ P $]\!] \Longrightarrow$ P
D_fresh_E1:
$[\![$ D_fresh c D; $\neg$ P $\Longrightarrow$ (c, t) $\in$ D $]\!] \Longrightarrow$ P
D_fresh_insert_E:
$[\![$ D_fresh c $\big($D $\cup$ $\{$(c', t)$\}\big)$; $[\![$ c $\neq$ c'; D_fresh c D $]\!] \Longrightarrow$ P $]\!] \Longrightarrow$ P

These lemmas provide natural-deduction-style introduction and elimination rules for D_fresh, which are more convenient in Isabelle/HOL proofs than unfolding the definition every time.

Lemma 5.18 (Properties of D_func)

D_func_empty_I:
D_func $\varnothing$
D_func_insert_I:
$[\![$ D_fresh c D; D_func D $]\!] \Longrightarrow$ D_func $\big($D $\cup$ $\{$(c, t)$\}\big)$

These two lemmas provide introduction rules for D_func.

This collection of lemmas represents only those properties that were actually needed; they are not intended as a complete axiomatization of the auxiliary predicates.

5.3 Evaluation

The evaluation judgment $S \triangleright K \vdash e \hookrightarrow w$ defined in Table 4.2 is represented by *two* inductively defined predicates: (eval s k e w) and (cont s k v w). The rules for eval

are named with an ev_ prefix, while the rules for cont have an op_ prefix, replacing the single prefix **ex_** used for the informal rules.

The first three arguments of eval and cont are considered inputs, while the fourth (the answer w) is considered an output. Evaluation statements will be viewed operationally, i.e., as resulting from a logic- programming-style proof search, starting from a conclusion with w unknown and working backwards using the inference rules to construct a derivation and determine the answer. The evaluation rules are written using reversed implication arrows ($\Longleftarrow$) to fit this viewpoint.

The answer (if any) is uniquely determined by the inputs, and is syntactically valid if the inputs are. However, these properties were not proved.

Definition 5.19 (eval, cont)

inductive eval :: $\lceil$ *state*, *cont*, *exp*, *final* $\rceil \Rightarrow$ *bool*
where ev_Zero_I: eval s k eZero w $\Longleftarrow$ cont s k eZero w
| ev_Suc_I: eval s k (eSuc e) w $\Longleftarrow$ eval s (kCons (λ v. eSuc v) k) e w
| ev_Pred_I: eval s k (ePred e) w $\Longleftarrow$ eval s (kPred k) e w
| ev_IsZero_I: eval s k (eIsZero e) w $\Longleftarrow$ eval s (kIsZero k) e w
| ev_True_I: eval s k eTrue w $\Longleftarrow$ cont s k eTrue w
| ev_False_I: eval s k eFalse w $\Longleftarrow$ cont s k eFalse w
| ev_IfThen_I: eval s k (eIfThen c tt ff) w $\Longleftarrow$ eval s (kIfThen tt ff k) c w
| ev_Unit_I: eval s k eUnit w $\Longleftarrow$ cont s k eUnit w
| ev_Pair_I: eval s k (ePair x y) w $\Longleftarrow$ eval s (kPair1 y k) x w
| ev_Fst_I: eval s k (eFst p) w $\Longleftarrow$ eval s (kFst k) p w
| ev_Snd_I: eval s k (eSnd p) w $\Longleftarrow$ eval s (kSnd k) p w
| ev_App_I: eval s k (f $ x) w $\Longleftarrow$ eval s (kArg x k) f w
| ev_Fn_I: eval s k (eFn F) w $\Longleftarrow$ $[\![$ abstr F; cont s k (eFn F) w $]\!]$
| ev_Fix_I: eval s k (eFix F) w $\Longleftarrow$ $[\![$ abstr F; eval s k (F (eFix F)) w $]\!]$
| ev_Letv_I: eval s k (eLetv v F) w $\Longleftarrow$ $[\![$ abstr F; eval s k (F v) w $]\!]$
| ev_Ref_I: eval s k (eRef e) w $\Longleftarrow$ eval s (kRef k) e w
| ev_Deref_I: eval s k (eDeref e) w $\Longleftarrow$ eval s (kDeref k) e w
| ev_Assign_I: eval s k (eAssign m e) w $\Longleftarrow$ eval s (kAssign1 e k) m w
| ev_Cell_I: eval s k (eCell c) w $\Longleftarrow$ cont s k (eCell c) w

and cont :: $\lbrack\, \mathit{state},\ \mathit{cont},\ \mathit{exp},\ \mathit{final} \,\rbrack \Rightarrow \mathit{bool}$
where op_Cons_I: cont s (kCons E k) v w $\Longleftarrow$ $[\![$ abstr E; cont s k (E v) w $]\!]$
| op_Arg_I: cont s (kArg x k) (eFn F) w $\Longleftarrow$ $[\![$ abstr F; eval s (kApp F k) x w $]\!]$
| op_App_I: cont s (kApp E k) v w $\Longleftarrow$ $[\![$ abstr E; eval s k (E v) w $]\!]$
| op_Pred_I1: cont s (kPred k) eZero w $\Longleftarrow$ cont s k eZero w
| op_Pred_I2: cont s (kPred k) (eSuc v) w $\Longleftarrow$ cont s k v w
| op_IsZero_I1: cont s (kIsZero k) eZero w $\Longleftarrow$ cont s k eTrue w
| op_IsZero_I2: cont s (kIsZero k) (eSuc v) w $\Longleftarrow$ cont s k eFalse w
| op_IfThen_I1: cont s (kIfThen tt ff k) eTrue w $\Longleftarrow$ eval s k tt w
| op_IfThen_I2: cont s (kIfThen tt ff k) eFalse w $\Longleftarrow$ eval s k ff w
| op_Pair1_I: cont s (kPair1 y k) v w $\Longleftarrow$ eval s (kPair2 v k) y w
| op_Pair2_I: cont s (kPair2 v k) v′ w $\Longleftarrow$ cont s k (ePair v v′) w
| op_Fst_I: cont s (kFst k) (ePair v v′) w $\Longleftarrow$ cont s k v w
| op_Snd_I: cont s (kSnd k) (ePair v v′) w $\Longleftarrow$ cont s k v′ w
| op_Ref_I: cont s (kRef k) v (fNew W)
$\Longleftarrow$ $[\![$ abstr W; $\bigwedge$ c. s_fresh c s $\Longrightarrow$ cont (sCons c v s) k (eCell c) (W c) $]\!]$
| op_Deref_I: cont s (kDeref k) (eCell c) w $\Longleftarrow$ $[\![$ s_lookup s c v; cont s k v w $]\!]$
| op_Assign1_I: cont s (kAssign1 e k) (eCell c) w $\Longleftarrow$ eval s (kAssign2 c k) e w
| op_Assign2_I: cont s (kAssign2 c k) v w $\Longleftarrow$ $[\![$ s_assign s c v s′; cont s′ k v w $]\!]$
| op_Done_I: cont s kDone v (fVal s v)

During the development of the formalization, an inductively defined predicate val was set up to recognize values. However, none of the evaluation rules (nor any other part of the formal theory) made use of it. Instead, they assume that an expression needs to be evaluated if passed to eval, while it is already a value if passed to cont. The latter is maintained as an invariant. On the other hand, when a value is substituted for a variable in an expression (e.g., in op_App_I), the fact that it is a value is forgotten, and it goes through the evaluation process again. That would be unacceptably inefficient behaviour in an implementation of Mini-ML, but since the end result is unchanged, it is not a problem for operational semantics.

In Chapter 4, primitives and functions from e are placed on the continuation while their arguments are evaluated, then brought back into e by the rule **ex_return**, where

another rule actually performs the operation. Here, after the arguments are evaluated by eval, the cont predicate takes over and immediately performs an operation taken from the end of the continuation, without bringing it back into e.

Each rule for eval applies to expressions built with a particular constructor, and is named correspondingly. The rules can be divided into several groups, according to how they correspond to the informal rules from Table 4.2:

Constants (ev_Zero_I, ev_True_I, ev_False_I, ev_Unit_I)

These rules just convert eval to cont, since the constants are already values. (There were no rules for constants in Table 4.2.)

Other syntactic values (ev_Cell_I, ev_Fn_I)

Memory references and lambda-abstractions are treated the same as constants, passing them immediately to cont. The latter rule has an abstr condition, as usual for Hybrid's HOAS.

Evaluating arguments (ev_Suc_I, ev_Pred_I, ev_IsZero_I, ev_IfThen_I, ev_Pair_I, ev_Fst_I, ev_Snd_I, ev_App_I, ev_Ref_I, ev_Deref_I, ev_Assign_I)

These rules work much like the corresponding rules from Table 4.2, passing one argument to eval as the expression to be evaluated and saving its context (including any other arguments) to the continuation. However, the saved context is represented using a specific constructor from Definition 5.5, rather than as a function expression. The one exception is ev_Suc_I, which uses the kCons constructor that does take a function expression (see op_Cons_I below).

Immediate substitutions (ev_Fix_I, ev_Letv_I)

These rules are identical in behaviour to the corresponding rules from Table 4.2, substituting an expression for a variable and passing the result back to eval, except that the variable binding is represented using Hybrid's HOAS.

Each rule for cont applies to continuations built with a particular constructor, and is named correspondingly. Once again, they correspond to informal rules from Table 4.2 in several different ways:

Substitution (op_Cons_I, op_App_I)

These rules are similar to **ex_return**, substituting the value into a function, except that here it is not a function expression but just the body of such an expression in its HOAS representation as a term of type $exp \Rightarrow exp$. The former rule passes the result to cont, the latter to eval. However, most instances of **ex_return** are folded into other rules, as described below.

Evaluating further arguments (op_Arg_I, op_Pair1_I, op_Assign1_I)

These rules combine **ex_return** with a rule to evaluate the next argument of an operation. That argument is taken from the continuation and passed to eval as the expression to be evaluated, while the just-evaluated value is saved to the continuation using a different constructor.

Performing an operation (op_Pred_I1, op_Pred_I2, op_IsZero_I1, op_IsZero_I2, op_IfThen_I1, op_IfThen_I2, op_Pair2_I, op_Fst_I, op_Snd_I, op_Deref_I, op_Assign2_I)

These rules combine **ex_return** with a rule that actually performs an operation. Where the result is known to be a value, it is passed to cont; otherwise, it is passed to eval.

The rules op_Deref_I and op_Assign2_I make use of auxiliary predicates defined earlier (s_lookup and s_assign) to manipulate the state.

Memory allocation (op_Ref_I)

This rule also performs an operation, but its formalization is somewhat trickier than the others. It passes (eCell c) to cont, for a universally quantified cell c, and

assigns the just-evaluated value to c in the state. It then abstracts c from the answer obtained under the universal quantifier, bringing it outside as a function W :: *cell* ⇒ *final* that is passed to fNew, producing a HOAS representation of a bound location variable.

This use of universal quantification is a HOAS technique that represents *free* location variables as Isabelle/HOL variables. The actual representation of the type *cell* is never used directly. It is only necessary that there are infinitely many distinct values of that type, so that there will be an instantiation of c with a fresh location.

It is also possible to instantiate c with locations that already occur in the state; such instances are excluded by the s_fresh condition. An invariant is maintained that any location occurring in the continuation, the current value, or a value in a memory cell must also occur in the state; this ensures that such locations are also excluded.

As future work, it should be possible to modify the definitions of s_lookup and s_assign, without changing their values on states satisfying s_func, so that they are preserved by (possibly non-injective) substitutions on location variables. This property would extend by induction to eval and cont, and the s_fresh condition could then be dropped as the non-fresh instances would be harmless.

Finishing up (op_Done_I)

This rule is identical in behaviour to **ex_init**, packaging the final state and value as the result of evaluation.

5.4 Typing judgments

The typing judgment $\Gamma \vdash e : \tau$ for expressions, defined in Table 4.1, is represented by an inductively defined predicate (ofe C D e t). As explained in Section 5.1, the

typing context Γ is split into two parts C and D, for program variables and location variables respectively.

Definition 5.20 (ofe)

inductive ofe :: $\big[(exp \times tp)\ set, (cell \times tp)\ set, exp, tp\big] \Rightarrow bool$
where ofe_Context_I: ofe C D x t $\Longleftarrow$ (x, t) ∈ C
| ofe_Zero_I: ofe C D eZero tNat
| ofe_Suc_I: ofe C D (eSuc e) tNat $\Longleftarrow$ ofe C D e tNat
| ofe_Pred_I: ofe C D (ePred e) tNat $\Longleftarrow$ ofe C D e tNat
| ofe_IsZero_I: ofe C D (eIsZero e) tBool $\Longleftarrow$ ofe C D e tNat
| ofe_True_I: ofe C D eTrue tBool
| ofe_False_I: ofe C D eFalse tBool
| ofe_IfThen_I: ofe C D (eIfThen c tt ff) t
$\Longleftarrow \llbracket$ ofe C D c tBool; ofe C D tt t; ofe C D ff t $\rrbracket$
| ofe_Unit_I: ofe C D eUnit tUnit
| ofe_Pair_I: ofe C D (ePair x y) (tPair t s) $\Longleftarrow \llbracket$ ofe C D x t; ofe C D y s $\rrbracket$
| ofe_Fst_I: ofe C D (eFst p) t $\Longleftarrow$ ofe C D p (tPair t s)
| ofe_Snd_I: ofe C D (eSnd p) s $\Longleftarrow$ ofe C D p (tPair t s)
| ofe_App_I: ofe C D (f $ x) s $\Longleftarrow \llbracket$ ofe C D f (tFun t s); ofe C D x t $\rrbracket$
| ofe_Fn_I: ofe C D (eFn F) (tFun t s)
$\Longleftarrow \llbracket$ abstr F; $\bigwedge$ x. ofe $\big(C \cup \{(x, t)\}\big)$ D (F x) s $\rrbracket$
| ofe_Fix_I: ofe C D (eFix F) t
$\Longleftarrow \llbracket$ abstr F; $\bigwedge$ x. ofe $\big(C \cup \{(x, t)\}\big)$ D (F x) t $\rrbracket$
| ofe_Letv_I: ofe C D (eLetv v F) t $\Longleftarrow \llbracket$ abstr F; ofe C D (F v) t $\rrbracket$
| ofe_Ref_I: ofe C D (eRef e) (tRef t) $\Longleftarrow$ ofe C D e t
| ofe_Deref_I: ofe C D (eDeref m) t $\Longleftarrow$ ofe C D m (tRef t)
| ofe_Assign_I: ofe C D (eAssign m e) t $\Longleftarrow \llbracket$ ofe C D m (tRef t); ofe C D e t $\rrbracket$
| ofe_Cell_I: ofe C D (eCell c) (tRef t) $\Longleftarrow$ (c, t) ∈ D

Most of these rules are semantically identical to the corresponding rules from Table 4.1. The rule ofe_Context_I generalizes **ofe_id** by allowing arbitrary expressions in place of program variables, and the rule ofe_Letv_I includes an abstr condition for its HOAS representation of a bound variable.

The most significant differences are found in the rules ofe_Fn_I and ofe_Fix_I. The corresponding rules **ofe_lam** and **ofe_fix** from Table 4.1 added a program variable and its type to the typing context. Here, the variable is represented by a universally quantified expression. (There is also an abstr condition as for ofe_Letv_I.) This is similar to the HOAS technique used for op_Ref_I, with the added benefit that the instances can be used to type the results of certain substitutions that will occur in Corollary 5.27.

It would be convenient to verify that the Hybrid term e :: *exp* fits the syntax of a Mini-ML expression while type-checking it. However, the predicate ofe fails to do so in two cases. One is ofe_Letv_I, which converts (eLetv v F) to (F v) without checking that v actually represents a Mini-ML value or even an expression – for example, v could be (c_ePair $$ eZero) and F could be (λ v. v $$ eZero). The other is ofe_Context_I, which deliberately does no checking so that ofe_Fn_I and ofe_Fix_I can use a simple universal quantification over *exp*.

The remaining typing judgments will also fail to verify syntactic validity, but only because of their dependence on ofe. As possible future work, relatively minor changes to the definition of ofe should allow all four typing judgments to imply syntactic validity of their subjects. However, for the purpose of subject reduction, this is not needed. (The problem of checking eLetv-expressions, at least, will be addressed in a later formalization in Definitions 7.9 and 7.10.)

The typing judgment $\Gamma \vdash K : \tau_1 \to \tau_2$ for continuations, defined in Table 4.3, is formalized as an inductively defined predicate (ofk C D k t r), in which the typing context Γ is split as before into C and D. While the form of the judgment corresponds to that of Table 4.3, though, the typing rules are quite different, as explained below.

Definition 5.21 (ofk)

inductive ofk :: $\big[(exp \times tp)\ set, (cell \times tp)\ set, cont, tp, tp\big] \Rightarrow bool$
where ofk_Cons_I: ofk C D (kCons E k) t r
$\Longleftarrow [\![$ abstr E; ofe C D (eFn E) (tFun t s); ofk C D k s r $]\!]$

```
| ofk_Arg_I: ofk C D (kArg x k) (tFun t s) r ⟸ ⟦ ofe C D x t; ofk C D k s r ⟧
| ofk_App_I: ofk C D (kApp E k) t r
                ⟸ ⟦ abstr E; ofe C D (eFn E) (tFun t s); ofk C D k s r ⟧
| ofk_Pred_I: ofk C D (kPred k) tNat r ⟸ ofk C D k tNat r
| ofk_IsZero_I: ofk C D (kIsZero k) tNat r ⟸ ofk C D k tBool r
| ofk_IfThen_I: ofk C D (kIfThen tt ff k) tBool r
                    ⟸ ⟦ ofe C D tt t; ofe C D ff t; ofk C D k t r ⟧
| ofk_Pair1_I: ofk C D (kPair1 y k) t r ⟸ ⟦ ofe C D y s; ofk C D k (tPair t s) r ⟧
| ofk_Pair2_I: ofk C D (kPair2 v k) s r ⟸ ⟦ ofe C D v t; ofk C D k (tPair t s) r ⟧
| ofk_Fst_I: ofk C D (kFst k) (tPair t s) r ⟸ ofk C D k t r
| ofk_Snd_I: ofk C D (kSnd k) (tPair t s) r ⟸ ofk C D k s r
| ofk_Ref_I: ofk C D (kRef k) t r ⟸ ofk C D k (tRef t) r
| ofk_Deref_I: ofk C D (kDeref k) (tRef t) r ⟸ ofk C D k t r
| ofk_Assign1_I: ofk C D (kAssign1 e k) (tRef t) r
                    ⟸ ⟦ ofe C D e t; ofk C D k t r ⟧
| ofk_Assign2_I: ofk C D (kAssign2 c k) t r ⟸ ⟦ (c, t) ∈ D; ofk C D k t r ⟧
| ofk_Done_I: ofk C D kDone r r
```

The rules ofk_Cons_I and ofk_App_I both correspond to **ofk_func**, the difference between kCons and kApp being irrelevant to typing, while ofk_Done_I corresponds to **ofk_init**. However, the representation of continuations as a datatype with many constructors (Definition 5.5) requires many corresponding rules here that did not appear in Chapter 4. These rules combine **ofk_func** with various typing rules for expressions, according to the correspondence between continuation constructors and function expressions.

For example, comparing **ex_pair** from Table 4.2 with ev_Pair_I from Definition 5.19, the continuation $K; \lambda x.\,(x, e_2)$ corresponds to (kPair1 y k), in which y represents e_2 and k represents K. The rule ofk_Pair1_I thus represents a combination of **ofk_func**, **ofe_lam**, **ofe_pair**, and **ofe_id**.

The typing judgment $\Delta \vdash S : \Delta'$ for states is formalized as an inductively defined predicate (ofs D s D′).

Definition 5.22 (ofs)

inductive ofs :: $\lfloor (cell \times tp)\ set, state, (cell \times tp)\ set \rfloor \Rightarrow bool$
where ofs_Nil_I: ofs D sNil $\varnothing$
| ofs_Cons_I: ofs D (sCons c v s) $\big(D' \cup \{(c, t)\}\big)$
$\Longleftarrow [\![$ ofs D s D'; s_fresh c s; D_fresh c D'; ofe $\varnothing$ D v t $]\!]$

These rules match the informal ones of Table 4.3 exactly, except for the addition of the freshness conditions (s_fresh c s) and (D_fresh c D′). Those conditions were implicit in the syntax in Chapter 4, but it is convenient here to fold them into ofs.

Lemma 5.23 (Basic properties of ofs)

ofs_s_func:
ofs D s D′ $\Longrightarrow$ s_func s
ofs_D_func:
ofs D s D′ $\Longrightarrow$ D_func D′
ofs_fresh_equiv:
ofs D s D′ $\Longrightarrow$ s_fresh c s = D_fresh c D′

The lemmas ofs_s_func and ofs_D_func extract the consequences of the freshness conditions from (ofs D s D′), namely that s and D′ are both valid in the sense of having at most one occurrence of any given location. The lemma ofs_fresh_equiv states that the same locations are found (or not found) in s and D′.

The typing judgment $\Delta \vdash w : \tau$ for answers is formalized as an inductively defined predicate (off C D w t), in which a context C of types for program variables has been added for more generality.

Definition 5.24 (off)

inductive off :: $\lfloor (exp \times tp)\ set, (cell \times tp)\ set, final, tp \rfloor \Rightarrow bool$
where off_Val_I: off C D (fVal s v) t $\Longleftarrow [\![$ ofs D s D; ofe C D v t $]\!]$
| off_New_I: off C D (fNew W) t
$\Longleftarrow [\![$ abstr W; $\bigwedge$ c. D_fresh c D
$\Longrightarrow$ off C $\big(D \cup \{(c, t')\}\big)$ (W c) t $]\!]$

The rule off_Val_I matches **off_loc** from Table 4.3, while off_New_I represents **off_new** using the same HOAS treatment of location variables that was used for op_Ref_I in Definition 5.19.

5.5 Semantic lemmas

Lemma 5.25 (Weakening properties)

ofe_C_weakening:
assumes ofe C D e t
shows $\bigwedge$ C′. C ⩽ C′ ⟹ ofe C′ D e t

ofe_D_weakening:
assumes ofe C D e t
shows $\bigwedge$ D′. D ⩽ D′ ⟹ ofe C D′ e t

ofk_D_weakening:
assumes ofk C D k t s
shows $\bigwedge$ D′. D ⩽ D′ ⟹ ofk C D′ k t s

ofs_D_weakening:
assumes ofs D s D′
shows $\bigwedge$ D″. D ⩽ D″ ⟹ ofs D″ s D′

These four lemmas show that adding elements to the contexts C and D does not invalidate typing statements. They are analogous to the weakening property in a sequent calculus. Weakening of C is used in proving the substitution property described below, while weakening of D is needed to carry typing statements through the allocation of a memory cell with op_Ref_I.

Lemma 5.26

ofe_cut:
assumes ofe $\big(\text{C} \cup \{(\text{x}, \text{s})\}\big)$ D e t **and** ofe C D x s
shows ofe C D e t

The lemma ofe_cut shows that an element (x, s) of the typing context (a *typing assumption*) may be dropped, strengthening the typing statement, if the expression

x has type s in the resulting, smaller context. (It is named by analogy with the cut rule in a sequent calculus, which will take its place in the two-level formalizations to follow.)

Proof. The conclusion is derived starting from a derivation of the first premise by removing the typing assumption (x, s) throughout, replacing each use of that assumption (via the axiom ofe_Context_I) with a derivation of the second premise, weakened as necessary.

More formally, we let $C_x = \big(C \cup \{(x, s)\}\big)$, and proceed by induction on the first premise, (ofe C_x D e t) (see Definition 5.20):

Case ofe_Context_I**:** In this case we have (e, t) $\in C_x$. If (e, t) $\in$ C, then the conclusion may likewise be derived by ofe_Context_I. Otherwise we must have (e, t) = (x, s), and the conclusion is identical to the second premise.

Case ofe_Fn_I**:** Here e is of the form eFn F where F :: *exp* $\Rightarrow$ *exp* and abstr F, t is of the form (tFun t_1 t_2) where t_1, t_2 :: *tp*, and the induction hypothesis is

$$\bigwedge y\ C'.\ [\![\ \big(C_x \cup \{(y, t_1)\}\big) = \big(C' \cup \{(x, s)\}\big);\ \text{ofe}\ C'\ D\ x\ s\ ]\!] \Longrightarrow \text{ofe}\ C'\ D\ (F\ y)\ t_2.$$

Let y :: *exp* be arbitrary, and $C_y = \big(C \cup \{(y, t_1)\}\big)$. We have $\big(C_x \cup \{(y, t_1)\}\big) = \big(C_y \cup \{(x, s)\}\big)$. We also have (ofe C_y D x s) by ofe_C_weakening (Lemma 5.25) applied to the second premise. Instantiating C' with C_y in the induction hypothesis, we deduce (ofe C_y D (F y) t_2). Finally, we generalize y and apply the rule ofe_Fn_I to conclude (ofe C D e t) as required.

Case ofe_Fix_I**:** This case is similar to ofe_Fn_I, except that t is not of the form (tFun t_1 t_2), and the other occurrences of t_1 and t_2 are replaced with t.

The remaining cases are proved trivially by induction, in each case applying the IH and then the same rule as the case being considered. □

Corollary 5.27

ofe_subst: $[\![\, \mathsf{ofe}\ \varnothing\ \mathrm{D}\ (\mathsf{eFn}\ \mathrm{F})\ (\mathsf{tFun}\ \mathrm{t}\ \mathrm{s});\ \mathsf{ofe}\ \varnothing\ \mathrm{D}\ \mathrm{e}\ \mathrm{t}\,]\!] \Longrightarrow \mathsf{ofe}\ \varnothing\ \mathrm{D}\ (\mathrm{F}\ \mathrm{e})\ \mathrm{s}$

Proof. Inverting Definition 5.20, the first premise must have been derived by ofe_Fn_I: the other rules give types for constructs other than eFn, except ofe_Context_I which does not apply here because the typing context for program variables is empty.

Thus, we must have $\big(\mathsf{ofe}\ \{(\mathrm{x},\ \mathrm{t})\}\ \mathrm{D}\ (\mathrm{F}\ \mathrm{x})\ \mathrm{s}\big)$ for all x :: *exp*. Instantiating x with e, and then applying the lemma ofe_cut (with the second premise) to remove (e, t) from the typing context, yields the desired conclusion (ofe $\varnothing$ D (F e) s). □

The corollary ofe_subst shows that substituting an expression of type t for the bound variable in the body of a function of type (tFun t s) yields a well-typed expression of type s. This is what is needed for type safety of β-reduction.

The proof illustrates an advantage of the HOAS encoding used for the typing rule ofe_Fn_I: we were able to obtain a typing statement for the result of β-reduction merely by instantiating a premise of that rule, with no need to explicitly reason about the substitution of a value for a variable in an expression.

It also illustrates a limitation of this encoding style: we must not allow a non-empty typing context C, because the generalized premise $\big(\mathsf{ofe}\ \mathrm{C}\ \mathrm{D}\ (\mathsf{eFn}\ \mathrm{F})\ (\mathsf{tFun}\ \mathrm{t}\ \mathrm{s})\big)$ could be derived by ofe_Context_I, and in that case the conclusion would not follow. However, since evaluation never proceeds under an fn binder (where the typing rules would generate a non-empty context), the less-general form is sufficient for proving type safety.

Lemma 5.28 (s_lookup_ofs)

$[\![\, \mathsf{s\_lookup}\ \mathrm{s}\ \mathrm{c}\ \mathrm{v};\ \mathsf{ofs}\ \mathrm{D}\ \mathrm{s}\ \mathrm{D}';\ (\mathrm{c},\ \mathrm{t}) \in \mathrm{D}'\,]\!] \Longrightarrow \mathsf{ofe}\ \mathrm{C}\ \mathrm{D}\ \mathrm{v}\ \mathrm{t}$

The lemma s_lookup_ofs proves what is needed for type safety of a dereference operation. That is, if we have a well-typed state s under a store typing D′, in which location c has type t, then the value v found at location c is indeed well-typed with

type t. This lemma is used only with D = D′, but the stronger statement is needed for proof by induction on the ofs premise. (It is also used only with C = ∅, and there is no real need for the more general statement with arbitrary C.)

Lemma 5.29 (s_assign_ofs)

⟦ s_assign s c v s′; ofs D s D′; (c, t) ∈ D′; ofe ∅ D v t ⟧ ⟹ ofs D s′ D′

The lemma s_assign_ofs proves what is needed for type safety of an assignment operation. That is, if we start with a well-typed state s under a store typing D′, in which location c has type t, and we assign a value v of type t to c to obtain a new state s′, then s′ is again well-typed under the same store typing D′. Again we will have D = D′ in use, the stronger statement being needed for induction.

5.6 Subject reduction

Theorem 5.30

subjRed: ⟦ eval s k e w; ofs D s D; ofe ∅ D e t; ofk ∅ D k t r ⟧ ⟹ off ∅ D w r

Proof. By induction on eval and cont (Definition 5.19), together with a corresponding property for cont:

⟦ cont s k v w; ofs D s D; ofe ∅ D v t; ofk ∅ D k t r ⟧ ⟹ off ∅ D w r

Case ev_Suc_I**:** In this case e is of the form (eSuc e′) for some e′ :: *exp*, and our induction hypothesis is the original statement for eval with e ↦ e′ and k ↦ (kCons (λ x. eSuc x) k):

⋀ D t. ⟦ ofs D s D; ofe ∅ D e′ t;
ofk ∅ D (kCons (λ x. eSuc x) k) t r ⟧ ⟹ off ∅ D w r

Its first premise is already available as the original ofs premise.

Inverting Definition 5.20, the ofe premise, (ofe ∅ D (eSuc e′) t), must have been derived by ofe_Suc_I; so we have t = tNat and (ofe ∅ D e′ tNat).

We derive the remaining premise of the induction hypothesis using the axioms and inference rules from Definitions 5.20 (ofe) and 5.21 (ofk). For any x :: *exp*, we have (ofe {(x, tNat)} D x tNat) by the axiom ofe_Context_I. Applying the rule ofe_Suc_I, we obtain (ofe {(x, tNat)} D (eSuc x) tNat). Next we apply the rule ofe_Fn_I, using the fact abstr (λ x. eSuc x), to obtain

(ofe $\varnothing$ D (fn x. eSuc x) (tFun tNat tNat)).

Finally, we apply the rule ofk_Cons_I to the latter statement, the same abstr fact, and the ofk premise to deduce $\big($ofk $\varnothing$ D (kCons (λ x. eSuc x) k) tNat r$\big)$.

The conclusion, (off $\varnothing$ D w r), follows by induction.

Case ev_Fix_I**:** Here e is of the form (eFix F) where F :: *exp* $\Rightarrow$ *exp* and (abstr F), and our induction hypothesis is the original statement for eval with e $\mapsto$ (F (eFix F)).

By inversion, the ofe premise, (ofe $\varnothing$ D (eFix F) t), must have been derived by ofe_Fix_I, so we have $\big($ofe {(x, t)} D (F x) t$\big)$ for all x :: *exp*. We instantiate x with (eFix F) to obtain $\big($ofe {(eFix F, t)} D (F (eFix F)) t$\big)$. Using the ofe premise a second time, we eliminate the typing assumption (eFix F, t) via ofe_cut (Lemma 5.26) to obtain $\big($ofe $\varnothing$ D (F (eFix F)) t$\big)$.

Since the remaining premises of the IH are the original ofs and ofk premises, the conclusion follows by induction.

Case op_Cons_I**:** Here k is of the form (kCons E k′) where E :: *exp* $\Rightarrow$ *exp*, (abstr E), and k′ :: *cont*. Our induction hypothesis is the original statement for cont with k $\mapsto$ k′ and v $\mapsto$ (E v).

Inverting Definition 5.21, the ofk premise, (ofk $\varnothing$ D (kCons E k′) t r), must have been derived by ofk_Cons_I, which means there exists t′ :: *tp* such that (ofe $\varnothing$ D (eFn E) (tFun t t′)) and (ofk $\varnothing$ D k′ t′ r). We combine the former with

the ofe premise using ofe_subst (Corollary 5.27) to obtain (ofe ∅ D (E v) t′). The conclusion then follows by induction.

Case op_App_I**:** This case is similar to op_Cons_I.

Case op_Ref_I**:** Here k is of the form (kRef k′) where k′ :: *cont*, and w is of the form (fNew W) where W :: *cell* ⇒ *final* and (abstr W). Our induction hypothesis is the original statement for cont with s ↦ (sCons c v s), k ↦ k′, v ↦ (eCell c), and w ↦ (W c), for any c :: *cell* satisfying (s_fresh c s):

$$\bigwedge c\ D\ t.\ [\![\, \mathsf{s\_fresh}\ c\ s;\ \mathsf{ofs}\ D\ (\mathsf{sCons}\ c\ v\ s)\ D;\ \mathsf{ofe}\ \varnothing\ D\ (\mathsf{eCell}\ c)\ t;\ \mathsf{ofk}\ \varnothing\ D\ k'\ t\ r\,]\!] \Longrightarrow \mathsf{off}\ \varnothing\ D\ (W\ c)\ r$$

Let c be an arbitrary location satisfying (D_fresh c D), and let s′ = (sCons c v s) and D′ = $\big(D \cup \{(c,\, t)\}\big)$. By ofs_fresh_equiv (Lemma 5.23), using the ofs premise, c also satisfies (s_fresh c s).

Using the weakening properties from Lemma 5.25, we obtain (ofe ∅ D′ v t) from the ofe premise (via ofe_D_weakening) and (ofs D′ s D) from the ofs premise (via ofs_D_weakening). We then apply the rule ofs_Cons_I (from Definition 5.22) to give (ofs D′ s′ D′), in which all three arguments include c. (The freshness assumptions are needed here.)

By inversion, the ofk premise, (ofk ∅ D (kRef k′) t r), must have been derived by ofk_Ref_I and we have (ofk ∅ D k′ (tRef t) r). Via ofk_D_weakening, this becomes (ofk ∅ D′ k′ (tRef t) r).

We have (ofe ∅ D′ (eCell c) (tRef t)) by the axiom ofe_Cell_I. We apply the induction hypothesis, instantiating D with D′ and t with (tRef t), to obtain (off ∅ D′ (W c) r).

Finally, we generalize c and apply the rule off_New_I (from Definition 5.24), which also needs (abstr W), to deduce the required conclusion

(off ∅ D (fNew W) r).

The remaining cases are straightforward combinations of inversion and application of rules for ofe and ofk ending with use of the induction hypothesis, except that op_Deref_I and op_Assign2_I need the lemmas s_lookup_ofs (Lemma 5.28) and s_assign_ofs (Lemma 5.29) respectively.

All cases except op_Ref_I were proved automatically by Isabelle/HOL, using a single automatic proof method except for specifying the needed lemmas for the cases op_Deref_I and op_Assign2_I. □

Chapter 6

Case Study: Two-level Approach

The first two-level formalization was done using a minimal intuitionistic specification logic. This specification logic was based on the one from `mlLang.thy` [67], with some small changes for better notation and proof automation.

Typing and evaluation judgments often include *logical* features, such as the use of contexts, that are not specific to the language they formalize. The idea of the two-level approach is to provide these features generically in the form of a *specification logic* (SL) in which particular object languages (OLs) can be encoded. Useful properties of the specification logic, such as derivability of a cut rule (which can be used to prove substitution properties for the OL), can be proved once and reused for many formalizations.

In this case, the main objective of using an intuitionistic SL is to represent typing contexts for program variables as logical assumptions. This change is intended to simplify the typing judgments, and allow some of the OL-specific lemmas of the previous chapter to be generalized to (reusable) properties of the SL. The results will be discussed following the definitions of the typing judgments.

For this and subsequent chapters, the term "meta-level" will refer to Isabelle/HOL, as distinguished from the specification logic. (Isabelle/HOL is itself a two-level

system built on top of Isabelle/Pure, but that will not be relevant here.) The term "object language" (OL) will refer to the language being formalized, i.e., Mini-ML with mutable references.

This formalization was constructed by modifying the one-level formalization from Chapter 5. The parts of the formal theory described in sections 5.1 (Syntax), 5.2 (Auxiliary predicates), and 5.3 (Evaluation) were reused without modification, except for the representation of typing assumptions for program variables in Section 5.1, which will be replaced by an SL representation in Section 6.2.

6.1 Specification logic

The specification logic is presented in a form optimized for proof search, with neither structural rules nor a cut rule as primitives. Instead, the logic's structural properties are built into its axiom rule, and then the structural rules and cut rule are established as derived rules.

The type *prp* represents logical formulas.

Definition 6.1 (SL formulas)

datatype a *prp* =

at a	(*notation* ⟨A⟩)
| true	(*notation* ⊤)
| conj (a *prp*) (a *prp*)	(*notation* B & C)
| imp a (a *prp*)	(*notation* A ⊃ B)
| forall (exp ⇒ a *prp*)	(*notation* all x. B)

The connectives of the specification logic consist of a truth constant (⊤), conjunction (B & C), implication (A ⊃ B) with an atomic premise, and universal quantification (all x. B) over Hybrid terms (intended to represent Mini-ML expressions). The type parameter a is the type of atomic formulas, which will be defined later as

a datatype with one constructor for each judgment to be represented in the SL. The constructor at ($\langle\_\rangle$) coerces atomic formulas to propositions.

Universal quantification is represented in HOAS style as a constructor forall with an argument of the functional type ($exp \Rightarrow a\ prp$). Unlike Hybrid's LAM :: $((a\ expr \Rightarrow a\ expr) \Rightarrow a\ expr)$ with its negative occurrence of the type to be defined, the form of HOAS used here is not so problematic and can be used in an Isabelle/HOL datatype.

The lack of Hybrid's abstr requirement means that the function argument of forall need not treat its *exp* argument generically, so it can represent infinite conjunctions, not just universal quantifications. However, in this formalization, all the functions actually used with forall will treat their arguments generically, so this potential extra expressive power is never used. As future work, another formalization could be attempted with SL formulas (and maybe even derivations) represented as Hybrid terms, and compared with this one.

The rules of the specification logic are given in the form of a sequent calculus. This consists of a predicate SL_entails relating a context (a list of atoms) to a conclusion (a formula). (Note that there is no real distinction between sequent calculus and natural deduction here, because there are neither left rules nor elimination rules.)

Definition 6.2 (SL sequent rules)

class ATM = TYPE + **fixes** prog :: $[\, a,\ a\ prp \,] \Rightarrow bool$ (*notation* A $\leftarrow$ B)
inductive SL_entails :: $[\, (a :: \text{ATM})\ list,\ a\ prp \,] \Rightarrow bool$ (*notation* $\Gamma \vdash$ B)
where bc: $\Gamma \vdash \langle \text{A} \rangle \Longleftarrow [\![\text{A} \leftarrow \beta;\ \Gamma \vdash \beta]\!]$
| ax: $\Gamma \vdash \langle \text{A} \rangle \Longleftarrow \text{A}$ mem Γ
| true_i: $\Gamma \vdash \top$
| conj_i: $\Gamma \vdash \text{B}\ \&\ \text{C} \Longleftarrow [\![\Gamma \vdash \text{B};\ \Gamma \vdash \text{C}]\!]$
| imp_i: $\Gamma \vdash \text{A} \supset \text{B} \Longleftarrow (\text{A}\ \#\ \Gamma \vdash \text{B})$
| all_i: $\Gamma \vdash$ all x. B x $\Longleftarrow (\bigwedge \text{x}.\ \Gamma \vdash \text{B x})$

Truth ($\top$) is provable by axiom, while the rule for conjunction (B & C) takes the two conjuncts (in the same context) as premises, as expected. The rule for implication

(A $\supset$ B) takes B as its premise, in a context augmented with A. The rule for universal quantification (all x. B x) takes as premises (B e) for all Hybrid terms e :: *exp*, where "for all" means universal quantification at the meta-level. This could be thought of as a form of HOAS, or simply as treating the SL's universal quantifier as an infinite conjunction.[1]

The predicate SL_entails converts an SL formula and context, which are *objects* of Isabelle/HOL, to a *statement* of Isabelle/HOL expressing SL provability of the formula in the context. This allows reasoning about specification-logic statements in the logic of Isabelle/HOL, including mixed SL and meta-level reasoning.

Atomic formulas are proved either by assumption from the context (using the rule ax), or by "backchaining" (using the rule bc). The latter rule supports logic-programming-style recursive specification of object-logic judgments using a predicate prog :: $\left[\, atm,\, prp \,\right] \Rightarrow bool$ (notation A $\leftarrow$ B), which is supplied by instantiating the type class ATM. The use of Isabelle/HOL definition mechanisms for prog makes the SL surprisingly flexible, with the ability to simulate some connectives that do not appear in Definition 6.1. Some examples of this flexibility will be seen when prog is actually defined for Mini-ML judgments, specifically in Definition 6.7.

This style of specification logic has been used in previous work with Hybrid [19, 21, 20], and it goes back to McDowell and Miller's work in $FO\lambda^{\Delta\mathbb{N}}$ [46, 47]. As compared with [21], our Definition 6.1 corresponds to the syntax of *goals*, and our Definition 6.2 includes both the syntax of *contexts* (represented here as lists) and the sequent rules. However, we do not fix a syntax for *clauses* to be followed in the definition of prog; instead we allow the full use of Isabelle/HOL definition mechanisms.

We do not include a natural-number argument as an induction measure, as the specification of SL statements as an inductively defined predicate already provides a

[1] Note that the SL's universal quantifier is not used to represent an object-level universal quantifier, but rather as part of the HOAS representation of certain typing rules (ofe_Fn_I and ofe_Fix_I) in Definition 6.7.

form of structural induction, which we will use in Section 6.3.

Lemma 6.3 (Structural properties)

SL_structural: $[\![G \vdash B; \forall a.\ a \mathsf{\ mem\ } G \longrightarrow a \mathsf{\ mem\ } D]\!] \Longrightarrow D \vdash B$
SL_weakening_1: $G \vdash B \Longrightarrow a \# G \vdash B$

The lemma SL_structural represents the usual three structural properties of intuitionistic sequent calculus: exchange, weakening, and contraction. These properties allow permuting formulas, inserting additional formulas, and removing duplicate formulas in the context of a sequent, respectively. They are combined here in the requirement that every assumption appearing in the original sequent appears also in the new sequent. The lemma SL_weakening_1 is the weakening property alone, in a form that is not the most general but will be convenient for later proofs.

Lemma 6.4

SL_cut: $[\![A \# G \vdash B; G \vdash \langle A \rangle]\!] \Longrightarrow G \vdash B$

The lemma SL_cut allows an atomic formula to be dropped from the context of a sequent if it can be proved in the SL from the rest of the context. It corresponds to the "cut-elimination" property of sequent calculi with a primitive cut rule. It is proved by replacing uses of the eliminated assumption A in the first premise with a derivation of the second premise, similarly to Lemma 5.26. (The fact that the context consists of atomic formulas greatly simplifies the proof.)

This lemma is closely connected with substitution properties such as Corollary 5.27; indeed, if explicit proof terms were defined for the specification logic, it would become a substitution property itself.

Lemma 6.5 (Simplifier rules)

SL_true_simp: $G \vdash \top = \mathsf{True}$
SL_conj_simp: $G \vdash B \,\&\, C = (G \vdash B \wedge G \vdash C)$
SL_all_simp: $G \vdash \mathsf{all}\ x.\ B\ x = (\forall x.\ G \vdash B\ x)$
SL_imp_simp: $G \vdash A \supset B = (A \# G \vdash B)$

These lemmas combine the SL's logical rules (i.e., the introduction rules for SL_entails) with their corresponding elimination rules in the form of equalities, for use as rewrite rules by Isabelle's simplifier. They are added to Isabelle's default simpset, and the intro/elim rules are added to the default claset, so that Isabelle's automatic proof methods can be used to reason in the SL.

Specification-level vs. meta-level representation

For each object-language judgment, there is a choice of whether to represent it in the specification logic, or directly as a meta-level predicate. If it is represented in the SL, then it can make use of the SL context and the general properties stated above; examples of these advantages will be seen later in this chapter. However, there are also some disadvantages: notational overhead, the need to combine the definitions into prog, harder-to-use induction rules, more unsafe intro/elim rules, etc.

Therefore, as a general guiding principle, if a particular judgment doesn't have a clear reason to be represented in the SL, then it should be represented at the meta-level instead. In this case study, following that principle has the additional benefit of testing the ability to mix SL and meta-level reasoning.

Similar considerations apply to Hybrid, perhaps even to a greater extent, since it provides forms of HOAS that otherwise would not work in Isabelle/HOL, yet it is untyped and lacks a good induction principle. This is why Hybrid was not used to represent SL formulas (or derivations) despite their use of HOAS.

Future work on reducing the disadvantages could shift the balance toward using specification logics (and Hybrid) more often.

6.2 Typing judgments

The typing judgments for expressions and continuations are represented as predicates of the SL, so the type of atomic formulas must have two corresponding constructors.

Definition 6.6

datatype *atm* =
  Ofe ((*cell* × *tp*) *set*) *exp tp*
| Ofk ((*cell* × *tp*) *set*) *cont tp tp*

The representation of typing contexts differs from Chapter 5 for program variables, whose types are given here by Ofe atoms in the SL context, replacing the explicit context argument C :: ((*exp* × *tp*) *set*) used in that chapter. For location variables, the context argument D :: ((*cell* × *tp*) *set*) from Chapter 5 is retained, and is used in SL atoms as well as meta-level predicates. The variables themselves are still represented as universally-quantified terms of types *exp* and *cell*.

An alternative will be seen in Chapters 7 to 9, where D is replaced with atoms in the SL context as well. That could be done in this chapter's intuitionistic SL also. However, this SL can express only monotonic constraints on its context, due to the weakening property (Lemma 6.3); e.g., an SL formula cannot express the fact that the context is empty, because any formula that is provable in the empty context is also provable in every other context by weakening. Since the typing judgment for states is not monotonic in its store-typing argument, it could not then be represented as an SL atom; so a meta-level predicate would have to be used instead, yet the store-typing argument would still be an SL context. Meta-level predicates on SL contexts will be seen in the following chapters (as "context invariants"), but they are a complicating factor, and for that reason D was kept as an explicit argument in this formalization.

The type class ATM from Definition 6.2 is instantiated for the type *atm* by supplying a prog predicate for that type. The rules for all atomic formulas must be given by this one predicate; however, for readability its definition will be split up

into two parts, representing the two typing judgments encoded in the SL. A trivial (base cases only) **inductive** definition is used, because that is more convenient than a direct definition using disjunction and existential quantification.

Definition 6.7 (Specification of SL atoms, part 1 – Ofe)

inductive prog_atm :: $[\, atm,\ atm\ prp \,] \Rightarrow bool$ $\quad$ ($notation$ A ← B)
where ofe_Zero_I: Ofe D eZero tNat ← ⊤
| ofe_Suc_I: Ofe D (eSuc e) tNat ← ⟨Ofe D e tNat⟩
| ofe_Pred_I: Ofe D (ePred e) tNat ← ⟨Ofe D e tNat⟩
| ofe_IsZero_I: Ofe D (eIsZero e) tBool ← ⟨Ofe D e tNat⟩
| ofe_True_I: Ofe D eTrue tBool ← ⊤
| ofe_False_I: Ofe D eFalse tBool ← ⊤
| ofe_IfThen_I: Ofe D (eIfThen c tt ff) t
← ⟨Ofe D c tBool⟩ & ⟨Ofe D tt t⟩ & ⟨Ofe D ff t⟩
| ofe_Unit_I: Ofe D eUnit tUnit ← ⊤
| ofe_Pair_I: Ofe D (ePair x y) (tPair t s) ← ⟨Ofe D x t⟩ & ⟨Ofe D y s⟩
| ofe_Fst_I: Ofe D (eFst p) t ← ⟨Ofe D p (tPair t s)⟩
| ofe_Snd_I: Ofe D (eSnd p) s ← ⟨Ofe D p (tPair t s)⟩
| ofe_App_I: Ofe D (f $ x) s ← ⟨Ofe D f (tFun t s)⟩ & ⟨Ofe D x t⟩
| ofe_Fn_I: Ofe D (eFn F) (tFun t s)
← all x. Ofe D x t ⊃ ⟨Ofe D (F x) s⟩ ⟸ abstr F
| ofe_Fix_I: Ofe D (eFix F) t
← all x. Ofe D x t ⊃ ⟨Ofe D (F x) t⟩ ⟸ abstr F
| ofe_Letv_I: Ofe D (eLetv v F) t ← ⟨Ofe D (F v) t⟩ ⟸ abstr F
| ofe_Ref_I: Ofe D (eRef e) (tRef t) ← ⟨Ofe D e t⟩
| ofe_Deref_I: Ofe D (eDeref m) t ← ⟨Ofe D m (tRef t)⟩
| ofe_Assign_I: Ofe D (eAssign m e) t ← ⟨Ofe D m (tRef t)⟩ & ⟨Ofe D e t⟩
| ofe_Cell_I: Ofe D (eCell c) (tRef t) ← ⊤ ⟸ (c, t) ∈ D

Many of the typing rules for expressions are similar to those of Definition 5.20, except that their premises are represented using SL connectives rather than Isabelle/HOL connectives, and the context argument C giving types for program variables has been removed.

The main differences are in the rules that deal with program variables, namely ofe_Fn_I and ofe_Fix_I. From an operational point of view (working backwards from the conclusion to type-check an expression), they use an SL implication to add an Ofe atom to the SL context, where the previous version added an element to the set C directly. Also, the expression substituted for the program variable is universally quantified in the SL, rather than at the meta-level. Finally, the axiom ofe_Context_I has been removed, since the use of assumptions is handled at the logical level by the axiom ax of the SL.

Several of these rules illustrate the flexibility of the SL's backchain rule for specifying OL judgments. Meta-level premises are mixed with SL premises in ofe_Fn_I, ofe_Fix_I, and ofe_Letv_I with their abstr conditions. Also, ofe_Cell_I has its only non-trivial premise at the meta-level. Existential quantification over types is simulated in ofe_App_I,

$$\bigwedge \text{f x s t. Ofe D (f \$ x) s} \leftarrow \langle \text{Ofe D f (tFun t s)} \rangle \;\&\; \langle \text{Ofe D x t} \rangle,$$

where the variable t :: *tp* appears only in the premises, allowing it to be instantiated arbitrarily without changing the conclusion.

Definition 6.8 (Specification of SL atoms, part 2 – Ofk)

inductive prog_atm :: $[\, atm,\ atm\ prp \,] \Rightarrow bool$ $\quad$ ($notation$ A ← B)
| ofk_Cons_I: Ofk D (kCons E k) t r
$\quad\quad \leftarrow \langle \text{Ofe D (eFn E) (tFun t s)} \rangle \;\&\; \langle \text{Ofk D k s r} \rangle \Longleftarrow \text{abstr E}$
| ofk_Arg_I: Ofk D (kArg x k) (tFun t s) r ← ⟨Ofe D x t⟩ & ⟨Ofk D k s r⟩
| ofk_App_I: Ofk D (kApp E k) t r
$\quad\quad \leftarrow \langle \text{Ofe D (eFn E) (tFun t s)} \rangle \;\&\; \langle \text{Ofk D k s r} \rangle \Longleftarrow \text{abstr E}$
| ofk_Pred_I: Ofk D (kPred k) tNat r ← ⟨Ofk D k tNat r⟩
| ofk_IsZero_I: Ofk D (kIsZero k) tNat r ← ⟨Ofk D k tBool r⟩
| ofk_IfThen_I: Ofk D (kIfThen tt ff k) tBool r
$\quad\quad \leftarrow \langle \text{Ofe D tt t} \rangle \;\&\; \langle \text{Ofe D ff t} \rangle \;\&\; \langle \text{Ofk D k t r} \rangle$
| ofk_Pair1_I: Ofk D (kPair1 y k) t r ← ⟨Ofe D y s⟩ & ⟨Ofk D k (tPair t s) r⟩
| ofk_Pair2_I: Ofk D (kPair2 v k) s r ← ⟨Ofe D v t⟩ & ⟨Ofk D k (tPair t s) r⟩

| ofk_Fst_I: Ofk D (kFst k) (tPair t s) r ← ⟨Ofk D k t r⟩
| ofk_Snd_I: Ofk D (kSnd k) (tPair t s) r ← ⟨Ofk D k s r⟩
| ofk_Ref_I: Ofk D (kRef k) t r ← ⟨Ofk D k (tRef t) r⟩
| ofk_Deref_I: Ofk D (kDeref k) (tRef t) r ← ⟨Ofk D k t r⟩
| ofk_Assign1_I: Ofk D (kAssign1 e k) (tRef t) r ← ⟨Ofe D e t⟩ & ⟨Ofk D k t r⟩
| ofk_Assign2_I: Ofk D (kAssign2 c k) t r ← ⟨Ofk D k t r⟩ ⟸ (c, t) ∈ D
| ofk_Done_I: Ofk D kDone r r ← ⊤

All of the typing rules for continuations match those of Definition 5.21 except that their premises are represented using SL atoms and connectives rather than predicates and Isabelle/HOL connectives.

The typing judgment for states only requires a store typing, not a typing context for program variables. Thus, following the general principle stated in Section 6.1, in this formalization it is represented at the meta-level as an inductively defined predicate (ofs D s D′).

Definition 6.9

inductive ofs :: $\big[(cell \times tp)\ set,\ state,\ (cell \times tp)\ set\big] \Rightarrow bool$
where ofs_Nil_I: ofs D sNil ∅
| ofs_Cons_I: ofs D (sCons c v s) $\big(D' \cup \{(c, t)\}\big)$
⟸ ⟦ ofs D s D′; s_fresh c s; D_fresh c D′; · ⊢ ⟨Ofe D v t⟩ ⟧

The typing rules for states are almost the same as those of Definition 5.22, except that the premise of ofs_Cons_I that involves typing of an expression is now an SL sequent. The basic properties of ofs from Lemma 5.23 were reused without modification, including their mostly-automatic proof methods, despite this change to the definition.

The typing judgment for answers is represented as an inductively defined predicate (off D w t). Unlike Definition 5.24, it does not have a typing context for program variables.

Definition 6.10

inductive off :: $\big[(cell \times tp)\ set, final, tp\big] \Rightarrow bool$
where off_Val_I: off D (fVal s v) t $\Longleftarrow \llbracket$ ofs D s D; $\cdot \vdash \langle$Ofe D v t$\rangle \rrbracket$
| off_New_I: off D (fNew W) t
$\Longleftarrow \llbracket$ abstr W; $\bigwedge$ c. D_fresh c D
$\Longrightarrow$ off $\big($D $\cup$ {(c, t′)}$\big)$ (W c) t $\rrbracket$

Other than the removal of the first argument C, the typing rules for answers differ from those of Definition 5.24 only in that the expression-typing premise of off_Val_I is now an SL sequent.

The typing judgment for answers in Chapter 5 did have a typing context for program variables. That suggests that it should be represented in the SL. However, its ofs premise would create a dependency loop, since ofs depends on SL_entails which depends in turn on prog. In a sense it would be a spurious one, because ofs really depends only on Ofe and Ofk; but Isabelle/HOL cannot track dependencies inside the SL.

One solution would be to put typing of states into the SL, even though it does not make use of any SL features. This does not cause any additional problems, indeed, it is possible to *define* a corresponding meta-level predicate equivalent to ofs from Definition 6.9. However, off was never used with a nonempty typing context for program variables in Chapter 5, so it is simpler just to remove that context and keep the judgment at the meta-level. (Alternative approaches will be seen in Chapter 8.)

6.3 Semantic lemmas and subject reduction

Essentially the same properties are needed here as in the previous formalization, but the use of a specification logic brings both advantages and disadvantages.

Lemma 6.11 (Substitution in Mini-ML functions)

ofe_subst:

$$[\![\cdot \vdash \langle \mathsf{Ofe}\ \mathrm{D}\ (\mathsf{eFn}\ \mathrm{F})\ (\mathsf{tFun}\ \mathrm{t}\ \mathrm{s}) \rangle;\ \mathrm{G} \vdash \langle \mathsf{Ofe}\ \mathrm{D}\ \mathrm{e}\ \mathrm{t} \rangle]\!] \Longrightarrow \mathrm{G} \vdash \langle \mathsf{Ofe}\ \mathrm{D}\ (\mathrm{F}\ \mathrm{e})\ \mathrm{s} \rangle$$

The substitution property ofe_subst is an easy corollary of SL_cut (Lemma 6.4), which is reusable for other object logics.

Of the weakening properties from Lemma 5.25, ofe_C_weakening requires no OL-specific counterpart here, as typing assumptions for program variables are just logical assumptions in the SL. Either SL_structural or its corollary SL_weakening_1 can be used directly. This is an advantage of the two-level approach for OLs that have contexts of assumptions.

On the other hand, we still have store typings as arguments of the typing judgments, and the op_Ref_I case of subject reduction will require weakening properties for them. However, the typing judgments for expressions and continuations are now represented in the SL, so we must replace the mostly-automatic induction proofs from the one-level formalization with a tricky induction on SL_entails. (An alternative approach that simplifies the proofs somewhat will be seen in Chapter 8, and compared with this approach.)

In particular, the desired statements must be strengthened for induction to a property of arbitrary sequents, and that requires defining new functions or relations to relate the premise to the weakened conclusion, where Lemma 5.25 could use $\mathrm{D} \leqslant \mathrm{D}'$.

Definition 6.12 (Weakening functions)

fun D_weaken_atm :: $[(cell \times tp)\ set,\ atm] \Rightarrow atm$
  where D_weaken_atm D′ (Ofe D e t) = Ofe (D ∪ D′) e t
  | D_weaken_atm D′ (Ofk D k t r) = Ofk (D ∪ D′) k t r

definition D_weaken_list D′ ≡ map (D_weaken_atm D′)

consts D_weaken_prp :: $[(cell \times tp)\ set,\ atm\ prp] \Rightarrow atm\ prp$
  primrec
  D_weaken_prp D′ ⟨A⟩ = ⟨D_weaken_atm D′ A⟩

D_weaken_prp D′ ⊤ = ⊤
D_weaken_prp D′ (B & C) = (D_weaken_prp D′ B & D_weaken_prp D′ C)
D_weaken_prp D′ (A ⊃ B) = (D_weaken_atm D′ A ⊃ D_weaken_prp D′ B)
D_weaken_prp D′ (forall F) = (all x. D_weaken_prp D′ (F x))

The functions D_weaken_atm, D_weaken_list, and D_weaken_prp replace the store typing argument of each occurrence of Ofe or Ofk with its union with a given set D′, in an SL atom, a list of atoms, or an SL proposition respectively.

Lemma 6.13 (Weakening properties)

SL_D_weakening:
assumes Γ ⊢ B **shows** D_weaken_list D′ Γ ⊢ D_weaken_prp D′ B
ofe_D_weakening:
assumes · ⊢ ⟨Ofe D e t⟩ **and** D ⩽ D′ **shows** · ⊢ ⟨Ofe D′ e t⟩
ofk_D_weakening:
assumes · ⊢ ⟨Ofk D k t r⟩ **and** D ⩽ D′ **shows** · ⊢ ⟨Ofk D′ k t r⟩
ofs_D_weakening:
assumes ofs D s D′ **and** D ⩽ D″ **shows** ofs D″ s D′

The lemma SL_D_weakening is the strengthened weakening property that is proved by induction on SL_entails. The lemmas ofe_D_weakening and ofk_D_weakening are its corollaries, corresponding to the lemmas of the same name in Chapter 5. The lemma ofs_D_weakening, which does not refer to the SL, is unchanged from Chapter 5.

Lemma 6.14 (s_lookup_ofs)

⟦ s_lookup s c v; ofs D s D′; (c, t) ∈ D′ ⟧ ⟹ · ⊢ ⟨Ofe D v t⟩

Lemma 6.15 (s_assign_ofs)

⟦ s_assign s c v s′; ofs D s D′; (c, t) ∈ D′; · ⊢ ⟨Ofe D v t⟩ ⟧ ⟹ ofs D s′ D′

The lemmas s_lookup_ofs and s_assign_ofs correspond directly to the lemmas of the same name in Chapter 5, with the only difference being the representation of typing for expressions.

Theorem 6.16 (Subject reduction)

$$[\![\, \mathsf{eval}\ \mathrm{s}\ \mathrm{k}\ \mathrm{e}\ \mathrm{w};\ \mathsf{ofs}\ \mathrm{D}\ \mathrm{s}\ \mathrm{D};\ \cdot \vdash \langle \mathsf{Ofk}\ \mathrm{D}\ \mathrm{k}\ \mathrm{t}\ \mathrm{r}\rangle;\ \cdot \vdash \langle \mathsf{Ofe}\ \mathrm{D}\ \mathrm{e}\ \mathrm{t}\rangle \,]\!] \Longrightarrow \mathsf{off}\ \mathrm{D}\ \mathrm{w}\ \mathrm{r}$$

Proof. By induction on eval and cont (as in Theorem 5.30), where the corresponding property for cont is

$$[\![\, \mathsf{cont}\ \mathrm{s}\ \mathrm{k}\ \mathrm{v}\ \mathrm{w};\ \mathsf{ofs}\ \mathrm{D}\ \mathrm{s}\ \mathrm{D};\ \cdot \vdash \langle \mathsf{Ofk}\ \mathrm{D}\ \mathrm{k}\ \mathrm{t}\ \mathrm{r}\rangle;\ \cdot \vdash \langle \mathsf{Ofe}\ \mathrm{D}\ \mathrm{v}\ \mathrm{t}\rangle \,]\!] \Longrightarrow \mathsf{off}\ \mathrm{D}\ \mathrm{w}\ \mathrm{r}.$$

Case op_Ref_I**:** In this case, k is of the form ($\mathsf{kRef}\ \mathrm{k}'$) where $\mathrm{k}' :: \mathit{cont}$, w is of the form ($\mathsf{fNew}\ \mathrm{W}$) where $\mathrm{W} :: \mathit{cell} \Rightarrow \mathit{final}$ and ($\mathsf{abstr}\ \mathrm{W}$), and our induction hypothesis is

$$\bigwedge \mathrm{c}\ \mathrm{D}\ \mathrm{t}.\ [\![\, \mathsf{s\_fresh}\ \mathrm{c}\ \mathrm{s};\ \mathsf{ofs}\ \mathrm{D}\ (\mathsf{sCons}\ \mathrm{c}\ \mathrm{v}\ \mathrm{s})\ \mathrm{D};\ \cdot \vdash \langle \mathsf{Ofk}\ \mathrm{D}\ \mathrm{k}'\ \mathrm{t}\ \mathrm{r}\rangle;\ \cdot \vdash \langle \mathsf{Ofe}\ \mathrm{D}\ (\mathsf{eCell}\ \mathrm{c})\ \mathrm{t}\rangle \,]\!] \Longrightarrow \mathsf{off}\ \varnothing\ \mathrm{D}\ (\mathrm{W}\ \mathrm{c})\ \mathrm{r}.$$

The proof of this case is similar to the corresponding case from Theorem 5.30, except that inversion now takes two steps. Starting from the Ofk premise, first we invert SL_entails ("$\vdash$", Definition 6.2) and deduce that the last SL rule used in its derivation must be bc, and thus we must have ($\mathsf{Ofk}\ \mathrm{D}\ (\mathsf{kRef}\ \mathrm{k}')\ \mathrm{t}\ \mathrm{r}$) $\leftarrow$ B and $\cdot \vdash$ B for some $\mathrm{B} :: \mathit{prp}$. Then we invert prog ("$\leftarrow$", Definition 6.8) and deduce that it must have been derived by ofk_Ref_I, so $\mathrm{B} = \langle \mathsf{Ofk}\ \mathrm{D}\ \mathrm{k}'\ (\mathsf{tRef}\ \mathrm{t})\ \mathrm{r}\rangle$.

Application of rules from Definitions 6.7 and 6.8 likewise requires two steps, first applying the SL rule bc, and then the specific typing rule in the form of an introduction rule for prog.

These differences are hidden in the formal proof text, because the two steps are combined into one using Isabelle/HOL's automatic proof methods. And otherwise the reasoning is identical to the one-level proof except for notational differences, specifically the use of SL sequent notation for typing statements for expressions and continuations.

The remaining cases are also similar to cases from Theorem 5.30, except that they were all proved by automatic methods, with only three of them needing special

treatment: a different automatic proof method was used for ev_Suc_I, and Lemmas 6.14 and 6.15 were given as rules for the auto method for op_Deref_I and op_Assign2_I respectively. □

Chapter 7

Case Study:
Linear Specification Logic

The second two-level formalization adds linear-logic features to the specification logic. The main objective of this change is to represent typing contexts for location variables as logical assumptions, and represent the typing judgment for states in the SL. This change is intended to further simplify the typing judgments and to remove some of the complications of mixing the use of the SL context with explicit context arguments (such as D in Definition 6.7).

This formalization was constructed by modifying the two-level formalization with intuitionistic SL from Chapter 6. The parts of the formal theory reused there from sections 5.1 (Syntax), 5.2 (Auxiliary predicates), and 5.3 (Evaluation) of the one-level formalization, were again reused without modification.

7.1 Specification logic

The specification logic presented here is from [46, § 5.2], except that as we did in Section 7.1, we allow an arbitrary Isabelle/HOL definition for prog in place of a specific

syntax of clauses, and we omit the natural-number argument used as an induction measure. We also include a multiplicative conjunction and truth constant, which deviates somewhat from the logic-programming style usually used in specification logics.

The type of logical formulas, *prp*, extends Definition 6.1 with new connectives:

Definition 7.1

datatype *a prp* =

at *a*	(*notation* ⟨A⟩)
| ttA	(*notation* ⊤)
| ttM	(*notation* **1**)
| conjA (*a prp*) (*a prp*)	(*notation* B & C)
| conjM (*a prp*) (*a prp*)	(*notation* B ⊗ C)
| impJ *a* (*a prp*)	(*notation* A ⊃ B)
| impL *a* (*a prp*)	(*notation* A ⊸ B)
| forall (*exp* ⇒ *a prp*)	(*notation* all x. B)

The linear SL has two truth constants (⊤ and **1**), two conjunctions (& and ⊗), and two implications (⊃ and ⊸). The differences between them will be explained along with the corresponding rules. The universal quantifier (all) is not duplicated.

SL statements are expressed using a predicate SL_entails on SL formulas that has two context arguments, an *intuitionistic context* Γ and a *linear context* Δ, both of type (*a* :: ATM) *list*. The former will admit the usual structural rules of exchange, weakening, and contraction, while the latter will admit only an exchange rule. (It will sometimes be convenient to refer to Γ and Δ together as "the context".)

The purpose of having two contexts is to support both intuitionistic specification as in Chapter 6 and specification using linear-logic features, and to allow them to be combined without the need for "modal" connectives as used in some variants of linear logic.

Definition 7.2 (SL sequent rules)

class ATM = TYPE + **fixes** prog :: $\lfloor a,\ a\ prp \rfloor \Rightarrow bool$ $\quad$ ($notation$ A $\leftarrow$ B)
inductive SL_entails :: $\lfloor (a :: \text{ATM})\ list,\ a\ list,\ a\ prp \rfloor \Rightarrow bool$
($notation$ Γ, $\Delta \vdash$ B)
where bc: Γ, $\Delta \vdash \langle \text{A} \rangle \Longleftarrow [\![\text{A} \leftarrow \text{B};\ \Gamma,\ \Delta \vdash \text{B}]\!]$
| axJ: Γ, $\cdot \vdash \langle \text{A} \rangle \Longleftarrow \text{A} \in \mathsf{set}\ \Gamma$
| axL: Γ, $[\,\text{A}\,] \vdash \langle \text{A} \rangle$
| ttA_i: Γ, $\Delta \vdash \top$
| ttM_i: Γ, $\cdot \vdash \mathbf{1}$
| conjA_i: Γ, $\Delta \vdash \text{B} \ \&\ \text{C} \Longleftarrow [\![\Gamma,\ \Delta \vdash \text{B};\ \Gamma,\ \Delta \vdash \text{C}]\!]$
| conjM_i: Γ, $\Delta \vdash \text{B} \otimes \text{C} \Longleftarrow [\![\Gamma,\ \Delta_{\text{L}} \vdash \text{B};\ \Gamma,\ \Delta_{\text{R}} \vdash \text{C};$
$\mathsf{mset}\ \Delta_{\text{L}} + \mathsf{mset}\ \Delta_{\text{R}} = \mathsf{mset}\ \Delta]\!]$
| impJ_i: Γ, $\Delta \vdash \text{A} \supset \text{B} \Longleftarrow \text{A} \#\ \Gamma,\ \Delta \vdash \text{B}$
| impL_i: Γ, $\Delta \vdash \text{A} \multimap \text{B} \Longleftarrow \Gamma,\ \text{A} \#\ \Delta \vdash \text{B}$
| all_i: Γ, $\Delta \vdash \mathsf{all}\ \text{x}.\ \text{B}\ \text{x} \Longleftarrow (\bigwedge \text{x}.\ \Gamma,\ \Delta \vdash \text{B}\ \text{x})$

There are two new distinctions in the linear SL as compared with the intuitionistic SL of Chapter 6, which are responsible for the splitting of connectives and rules. The first affects rules that directly involve the context (ax and imp_i): it is necessary to specify which context to use, intutionistic or linear, and this is indicated by a suffix of J or L respectively.

The second distinction affects rules that have more than one premise (conj_i) or no premises (tt_i). The linear context can either be treated *additively*, requiring all premises and the conclusion to agree, or *multiplicatively*, allowing the premises to have separate linear contexts and merging them for the conclusion (with duplicates retained). (See Girard, [26].) This distinction is indicated by suffixes of A or M.

The rules for backchaining (bc) and universal quantification (all_i) are affected by neither distinction, so they are reused from Definition 6.2 with the only change being the use of two context arguments for each SL statement.

As in Chapter 6, the desired structural properties are built into the axiom rules. This means that those rules allow arbitrary formulas in the intuitionistic context (in

addition to the formula A in the case of axJ), but require the linear context to be empty for axJ and to consist of the single atomic formula A for axL.

The additive vs. multiplicative distinction could be applied also to the intutionistic context, but the intended structural properties would make it irrelevant. However, to avoid the need for explicit structural rules, that context is always treated additively (as it was in Chapter 6). For the same reason, the rule conjM_i builds in the exchange property by using a multiset equality rather than a list equality to merge the linear contexts from the premises.

When the linear context Δ is empty, it will usually be omitted.

Lemma 7.3

SL_structural: $[\![\Gamma, \Delta \vdash B;\ \mathsf{set}\ \Gamma \subseteq \mathsf{set}\ \Gamma';\ \mathsf{mset}\ \Delta = \mathsf{mset}\ \Delta']\!] \Longrightarrow \Gamma', \Delta' \vdash B$

The lemma SL_structural proves that the SL as defined here achieves the intended structural properties stated prior to Definition 7.2: exchange, weakening, and contraction for Γ, and exchange for Δ. All of these properties are combined in the set inclusion and multiset equality premises of the lemma.

The intuitionistic vs. linear distiction applies to the (derived) cut rule as well, so we have two lemmas SL_cutJ and SL_cutL.

Lemma 7.4

SL_cutJ: $[\![A \# \Gamma, \Delta \vdash B;\ \Gamma \vdash \langle A \rangle]\!] \Longrightarrow \Gamma, \Delta \vdash B$

SL_cutL: $[\![\Gamma, A \# \Delta_1 \vdash B;\ \Gamma, \Delta_2 \vdash \langle A \rangle;$
$\mathsf{mset}\ \Delta_1 + \mathsf{mset}\ \Delta_2 = \mathsf{mset}\ \Delta]\!] \Longrightarrow \Gamma, \Delta \vdash B$

In the case of a cut on a formula in the intuitionistic context (SL_cutJ), that formula should be provable from the remaining formulas in the intuitionistic context, with an empty linear context.

For a cut on a formula in the linear context (SL_cutL), the proof of that formula may have its own linear context, which is combined multiplicatively with the linear context from the first premise (excluding the cut formula) to form the linear context

for the conclusion. This is expressed using a multiset-equality premise that builds in the exchange property, as it was for conjM_i, to facilitate backward reasoning.

Lemma 7.5

conjA_e: $\Gamma, \Delta \vdash B \,\&\, C \Longrightarrow \Gamma, \Delta \vdash B \qquad \Gamma, \Delta \vdash B \,\&\, C \Longrightarrow \Gamma, \Delta \vdash C$
impJ_e: $[\![\Gamma, \Delta \vdash A \supset B; \Gamma \vdash \langle A \rangle]\!] \Longrightarrow \Gamma, \Delta \vdash B$
impL_e: $[\![\Gamma, \Delta_1 \vdash A \multimap B; \Gamma, \Delta_2 \vdash \langle A \rangle]\!] \Longrightarrow \Gamma, \Delta_1 \mathbin{@} \Delta_2 \vdash B$
all_e: $\Gamma, \Delta \vdash \mathsf{all}\ x.\ B_p\ x \Longrightarrow \Gamma, \Delta \vdash B_p\ x$

The lemmas and_e, imp_e, impL_e, and all_e are derived SL rules that allow natural-deduction-style elimination for most of the SL connectives. The one omission is the multiplicative conjunction ($\otimes$), for which the usual approach to elimination involves putting both conjuncts into the linear context, which is incompatible with the restriction of contexts to atomic formulas only. (For the same reason, sequent-style left rules cannot even be stated for this SL.)

Definition 7.6

$\mathsf{J}\ B = (B \,\&\, \mathbf{1}) \otimes \top$

The derived connective J is used to convert instances of intuitionistic SL predicates – i.e., predicates that make no use of the linear context and expect it to be empty – into a form that ignores the linear context. This property is formalized by the lemma SL_J_simp:

Lemma 7.7 (SL_J_simp)

$(\Gamma, \Delta \vdash \mathsf{J}\ B) = (\Gamma \vdash B)$

An alternative approach would be to modify the axiom axJ to allow a non-empty linear context. Then if intuitionistic SL predicates were built using $\top$ for truth, and not allowed to appear as assumptions in the linear context, they would ignore the linear context by default.

7.2 Typing judgments, etc.

In this formalization, SL atoms are used for many different predicates and context elements. The type of atomic formulas has eight constructors, as compared with just two (Ofe and Ofk) in Definition 6.6.

Definition 7.8

datatype *atm* = IsTerm *exp* | Ofe *exp tp* | Ofk *cont tp tp* | Ofc *cell tp*
| OfcL *cell tp* | OfsL *state* | OffL *final tp* | OffNewL *cell final tp*

In accordance with the stated objective of using a linear SL, the entire typing context is represented in the SL context, so there are no explicit context arguments. Types for program variables are specified by Ofe atoms in the intuitionistic context, while types for location variables are specified twice, by Ofc atoms in the intuitionistic context and by OfcL atoms in the linear context. (These correspond to the two occurrences of the store typing, Δ and Δ', in the typing judgment for states given in Table 4.3.) Ofe is also a judgment with introduction rules, while Ofc and OfcL are only used as context elements and have no introduction rules. These atoms (and IsTerm, discussed after Definition 7.9) are the only ones permitted to occur in the SL context. (The structure of SL contexts will be formalized as a *context invariant* in Section 7.3.)

There is also a distinction (although it is not formalized) between the atoms with L suffixes, which make use of the linear context, and those without such suffixes, which are purely intuitionistic and mostly ignore the linear context. (The exception is when such atoms appear in the intuitionistic context, in which case the linear context must be empty to apply axJ; for this reason, sequents with these atoms on the right-hand side will always have empty linear contexts.)

The intuitionistic representations of typing for expressions (Ofe) and continuations (Ofk) are retained from Chapter 6, while the linear features of the SL are used in typing for states (OfsL) and answers (OffL). The atom OffNewL is an auxiliary

predicate used by OffL to relativize a universal quantification over locations. The atom IsTerm checks syntactic validity of expressions.

Definition 7.9 (Specification of SL atoms, part 1 of 5 – Ofe)

inductive prog_atm :: $\left[\, atm,\ atm\ prp \,\right] \Rightarrow bool$ $\quad$ (*notation* A ← B)
where ofe_Zero_I: Ofe eZero tNat ← ⊤
| ofe_Suc_I: Ofe (eSuc e) tNat ← ⟨Ofe e tNat⟩
| ofe_Pred_I: Ofe (ePred e) tNat ← ⟨Ofe e tNat⟩
| ofe_IsZero_I: Ofe (eIsZero e) tBool ← ⟨Ofe e tNat⟩
| ofe_True_I: Ofe eTrue tBool ← ⊤
| ofe_False_I: Ofe eFalse tBool ← ⊤
| ofe_IfThen_I: Ofe (eIfThen c tt ff) t
← ⟨Ofe c tBool⟩ & ⟨Ofe tt t⟩ & ⟨Ofe ff t⟩
| ofe_Unit_I: Ofe eUnit tUnit ← ⊤
| ofe_Pair_I: Ofe (ePair x y) (tPair t s) ← ⟨Ofe x t⟩ & ⟨Ofe y s⟩
| ofe_Fst_I: Ofe (eFst p) t ← ⟨Ofe p (tPair t s)⟩
| ofe_Snd_I: Ofe (eSnd p) s ← ⟨Ofe p (tPair t s)⟩
| ofe_App_I: Ofe (f $ x) s ← ⟨Ofe f (tFun t s)⟩ & ⟨Ofe x t⟩
| ofe_Fn_I: Ofe (eFn F) (tFun t s)
← all x. Ofe x t ⊃ ⟨Ofe (F x) s⟩ ⟸ abstr F
| ofe_Fix_I: Ofe (eFix F) t
← all x. Ofe x t ⊃ ⟨Ofe (F x) t⟩ ⟸ abstr F
| ofe_Letv_I: Ofe (eLetv v F) t
← ⟨IsTerm (eLetv v F)⟩ & ⟨Ofe (F v) t⟩ ⟸ abstr F
| ofe_Ref_I: Ofe (eRef e) (tRef t) ← ⟨Ofe e t⟩
| ofe_Deref_I: Ofe (eDeref m) t ← ⟨Ofe m (tRef t)⟩
| ofe_Assign_I: Ofe (eAssign m e) t ← ⟨Ofe m (tRef t)⟩ & ⟨Ofe e t⟩
| ofe_Cell_I: Ofe (eCell c) (tRef t) ← ⟨Ofc c t⟩

The typing rules for expressions are identical to those of Definition 6.7, except for the removal of the context argument D, the use of IsTerm in the rule ofe_Letv_I, and the rule ofe_Cell_I which now obtains a type for the location c from the intuitionistic context using Ofc, where the previous formalization obtained it from D.

For this formalization, the typing judgments were set up to imply syntactic validity of their subjects, so long as the same property holds for the context. This addresses the issue noted after Definition 5.20. In the case of ofe_Letv_I, an auxiliary SL predicate IsTerm was needed to achieve this property, since a rule similar to ofe_Fn_I (using HOAS based on Ofe atoms in the context) would reject ML-polymorphic eLetv expressions.

Definition 7.10 (Specification of SL atoms, part 2 of 5 – IsTerm)

inductive prog_atm :: $\left[\, atm,\ atm\ prp \,\right] \Rightarrow bool$ $\quad$ ($notation$ A $\leftarrow$ B)
where isterm_Ofe_I: IsTerm e $\leftarrow$ ⟨Ofe e t⟩
| isterm_App_I: IsTerm (f $ x) $\leftarrow$ ⟨IsTerm f⟩ & ⟨IsTerm x⟩
| isterm_Fn_I: IsTerm (eFn F)
$\leftarrow$ all x. IsTerm x $\supset$ ⟨IsTerm (F x)⟩ $\Longleftarrow$ abstr F
| isterm_Fix_I: IsTerm (eFix F)
$\leftarrow$ all x. IsTerm x $\supset$ ⟨IsTerm (F x)⟩ $\Longleftarrow$ abstr F
| isterm_Letv_I: IsTerm (eLetv v F)
$\leftarrow$ ⟨IsTerm v⟩ & (all x. IsTerm x $\supset$ ⟨IsTerm (F x)⟩) $\Longleftarrow$ abstr F
| ...

The non-HOAS inductive cases are all straightforward, and only isterm_App_I is shown as a representative example. The HOAS inductive cases resemble the corresponding cases for Ofe with types elided, except for isterm_Letv_I which checks v and F separately where ofe_Letv_I could not.

No base cases are necessary as they are covered by isterm_Ofe_I. In the case of eCell, the combination of isterm_Ofe_I and ofe_Cell_I checks the location against the context, which excludes syntactically invalid terms such as

eLetv (eLetv eUnit (λ x. eCell x)) (λ y. eUnit),

in which a bound program variable (x) appears in a context expecting a location, but the offending subterm is never type-checked because it is located in the first argument of an eLetv whose bound variable (y) does not occur in its body.

The rule isterm_Ofe_I is also essential for IsTerm to work properly in ofe_Letv_I, where there may be universally quantified Hybrid terms with corresponding Ofe atoms in the context, introduced by ofe_Fn_I or ofe_Fix_I and representing bound program variables. These terms must be treated as valid expressions even though some instances will not fit the syntax of Mini-ML expressions.

Since isterm_Ofe_I matches IsTerm applied to an arbitrary expression, it complicates inversion of the definition. However, that kind of reasoning will not be required for IsTerm.

An alternative approach would be to define IsTerm separately from Ofe, with its own base cases in place of isterm_Ofe_I, and instead modify Ofe to represent bound variables using both Ofe and IsTerm atoms in the context. This approach would support simpler inversion for IsTerm; but it would not be a good choice here, since it would instead complicate reasoning about Ofe, which will be used extensively. It would also fail to ensure the syntactic validity of expressions that pass type-checking, unless combined with other changes. (One sufficient change, which is planned as future work in any case, would be to introduce a constructor for locations to eliminate their overlap with the other syntactic classes.)

Definition 7.11 (Specification of SL atoms, part 3 of 5 – Ofk)

inductive prog_atm :: $\left[\,atm,\ atm\ prp\,\right] \Rightarrow bool$ $\quad$ (*notation* A ← B)
where ofk_Cons_I: Ofk (kCons E k) t r
$\qquad$ ← ⟨Ofe (eFn E) (tFun t s)⟩ & ⟨Ofk k s r⟩ ⟸ abstr E
| ofk_Arg_I: Ofk (kArg x k) (tFun t s) r ← ⟨Ofe x t⟩ & ⟨Ofk k s r⟩
| ofk_App_I: Ofk (kApp E k) t r
$\qquad$ ← ⟨Ofe (eFn E) (tFun t s)⟩ & ⟨Ofk k s r⟩ ⟸ abstr E
| ofk_Pred_I: Ofk (kPred k) tNat r ← ⟨Ofk k tNat r⟩
| ofk_IsZero_I: Ofk (kIsZero k) tNat r ← ⟨Ofk k tBool r⟩
| ofk_IfThen_I: Ofk (kIfThen tt ff k) tBool r
$\qquad$ ← ⟨Ofe tt t⟩ & ⟨Ofe ff t⟩ & ⟨Ofk k t r⟩
| ofk_Pair1_I: Ofk (kPair1 y k) t r ← ⟨Ofe y s⟩ & ⟨Ofk k (tPair t s) r⟩
| ofk_Pair2_I: Ofk (kPair2 v k) s r ← ⟨Ofe v t⟩ & ⟨Ofk k (tPair t s) r⟩

| ofk_Fst_I: Ofk (kFst k) (tPair t s) r ← ⟨Ofk k t r⟩
| ofk_Snd_I: Ofk (kSnd k) (tPair t s) r ← ⟨Ofk k s r⟩
| ofk_Ref_I: Ofk (kRef k) t r ← ⟨Ofk k (tRef t) r⟩
| ofk_Deref_I: Ofk (kDeref k) (tRef t) r ← ⟨Ofk k t r⟩
| ofk_Assign1_I: Ofk (kAssign1 e k) (tRef t) r ← ⟨Ofe e t⟩ & ⟨Ofk k t r⟩
| ofk_Assign2_I: Ofk (kAssign2 c k) t r ← ⟨Ofc c t⟩ & ⟨Ofk k t r⟩
| ofk_Done_I: Ofk kDone r r ← ⊤

The typing rules for continuations are identical to those of Definition 6.8, except for the removal of the context argument D.

Definition 7.12 (Specification of SL atoms, part 4 of 5 – OfsL)

inductive prog_atm :: [*atm*, *atm prp*] ⇒ *bool* (*notation* A ← B)
where ofsL_Nil_I: OfsL sNil ← **1**
| ofsL_Cons_I: OfsL (sCons c v s)
← ⟨OfcL c t⟩ ⊗ (J ⟨Ofe v t⟩ & ⟨OfsL s⟩) ⟸ s_fresh c s

The typing rules for states correspond to Definition 6.9, but since store typings have been moved to the SL context, they must be manipulated using SL connectives rather than explicit list operations – a process that is best explained from an operational point of view. The multiplicative truth constant **1** serves as a base case, requiring the linear context to be empty. For the recursive case, the multiplicative conjunction ⊗ is used to match and remove an OfcL atom from the linear context, while J and the additive conjunction & are used to type-check v using the intuitionistic SL predicate Ofe without affecting the linear context.

Definition 7.13 (Specification of SL atoms, part 5 of 5 – OffL, OffNewL)

inductive prog_atm :: [*atm*, *atm prp*] ⇒ *bool* (*notation* A ← B)
where offL_Val_I: OffL (fVal s v) t ← ⟨OfsL s⟩ & J ⟨Ofe v t⟩
| offL_New_I: OffL (fNew W) t ← all c. ⟨OffNewL c (W c) t⟩ ⟸ abstr W
| offNewL_I1: OffNewL c w t ← J ⟨Ofc c t′⟩
| offNewL_I: OffNewL c w t ← (Ofc c t′ ⊃ (OfcL c t′ ⊸ ⟨OffL w t⟩))

The typing rules for answers correspond to Definition 6.10, with some complications caused by the freshness condition (D_fresh c D) found there. The corresponding condition here is that the location c does not occur in an Ofc atom in the context. This condition must not be represented by an SL atom on the left-hand side of an implication, since that would put it into the context, where there are no SL rules that would allow it to function as a defined predicate.

Instead, we use a classical-logic equivalence to transform the implication from Definition 6.10 into a disjunction with the freshness condition negated. The latter is represented by the premise J ⟨Ofc c t′⟩ of offNewL_I1. The derived connective J is used to make this rule applicable regardless of the linear context, and existential quantification of t′ is simulated by the same method used for ofe_App_I.

The disjunction is also simulated, by defining an auxiliary SL predicate OffNewL with a rule for each disjunct. These rules share the same conclusion, a generic instance of OffNewL, unlike most of the other SL predicates whose rules have disjoint patterns as their conclusions. (This technique is another example of the flexibility of the backchaining rule as mentioned in Section 6.1.)

The premise of offNewL_I corresponds to the recursive call of off in Definition 6.10, in which an element is added to the store typing. Since the store typing is now represented by Ofc and OfcL atoms in the SL context, adding an element is done using SL implications.

Operationally, statements of the form Γ, Δ ⊢ (OffL (fNew W) t) are proved by backwards application of the rules bc, offL_New_I, and all_i, followed by a case distinction on whether the universally quantified location c occurs (as an atom (Ofc c t′) for some t :: *tp*) in the context. The rule offNewL_I1 eliminates the case where it does occur, while the other case provides a freshness condition to be used together with offNewL_I. (This freshness condition will take the form of a predicate Ofc_fresh defined in Definition 7.17.)

7.3 Context invariants

Since the SL context can hold arbitrary atoms, including those that represent object-language judgments, it is essential that it be used in a controlled way. The typing rules defined in Section 7.2 maintain particular invariants on the context, e.g., that the linear context consists only of OfcL atoms. To prove properties of the typing judgments, we must formalize these *context invariants*.

Since the evaluation strategy of our object language never evaluates under an fn or fix binder, for the purpose of subject reduction we will not encounter contexts containing Ofe atoms. (This is the reason why we had *empty* contexts in Chapter 6.)

Definition 7.14

ctxt_invar Γ Δ =
  ($\forall$ A $\in$ set Γ. ctxt_elt_J A Γ Δ) $\wedge$ ($\forall$ A $\in$ mset Δ. ctxt_elt_L A Γ Δ)

The context invariant ctxt_invar is defined in terms of conditions on the individual atoms in the context, ctxt_elt_J for the intuitionistic context and ctxt_elt_L for the linear context. These predicates also take the entire SL context as the arguments Γ and Δ.

Definition 7.15

ctxt_elt_J A Γ Δ =
  ($\exists$ c t. A = Ofc c t $\wedge$ Ofc_mem1 c t Γ $\wedge$ OfcL_mem1 c t Δ)
ctxt_elt_L A Γ Δ =
  ($\exists$ c t. A = OfcL c t $\wedge$ Ofc_mem1 c t Γ $\wedge$ OfcL_mem1 c t Δ)

Each atom in the intuitionistic context is required to be of the form Ofc c t and to have a matching atom OfcL c t in the linear context. Similarly, each atom in the linear context is required to be of the form OfcL c t and to have a matching Ofc atom in the intuitionistic context. Also, each location c must occur in at most one Ofc atom and one OfcL atom.

The predicates Ofc_mem1 and OfcL_mem1 check for both the presence of an appropriate atom and its uniqueness for the location c.

Definition 7.16

Ofc_mem1 c t Γ =
  Ofc c t ∈ set Γ ∧ Ofc_fresh c (remove1 (Ofc c t) Γ)
OfcL_mem1 c t Δ =
  OfcL c t ∈ mset Δ ∧ OfcL_fresh c (remove1 (OfcL c t) Δ)

(The function remove1 takes two arguments x :: *a* and L :: (*a list*), and its value is the list L with the first occurrence of x, if any, removed.)

The predicate Ofc_fresh serves as the freshness condition for bound location variables introduced with fNew, as mentioned after Definition 7.13.

Definition 7.17

Ofc_fresh c Γ = (∀ c′ t′. Ofc c′ t′ ∈ set Γ ⟶ c′ ≠ c)
OfcL_fresh c Δ = (∀ c′ t′. OfcL c′ t′ ∈ mset Δ ⟶ c′ ≠ c)

There is a second, weaker version of the context invariant called ctxt_invar1:

Definition 7.18

ctxt_invar1 Γ Δ =
  (∀ A ∈ set Γ. ctxt_elt_J1 A Γ) ∧ (∀ A ∈ mset Δ. ctxt_elt_L A Γ Δ)
ctxt_elt_J1 A Γ =
  (∃ c t. A = Ofc c t ∧ Ofc_mem1 c t Γ)

It differs from ctxt_invar in that Ofc atoms are not required to have matching OfcL atoms, so the set of locations given types by the linear context may be a subset of the locations found in the unresticted context.

This variant is needed for proving properties of Ofs, since that typing judgment removes OfcL atoms from the context as they are checked. (The stronger version ctxt_invar is only truly necessary for the op_Assign2_I case of subject reduction, which deals with assignment of a new value to a memory cell.)

Lemma 7.19 (SL_atom_ctxt1_inv)

$$\begin{array}{l}[\![\,\Gamma,\,\Delta \vdash \langle \mathrm{A}\rangle;\ \mathsf{ctxt\_invar1}\ \Gamma\ \Delta\,]\!] \Longrightarrow \\ \quad \big(\exists\ \mathrm{c}\ \mathrm{t}.\ (\mathrm{A} = \mathsf{Ofc}\ \mathrm{c}\ \mathrm{t}) \wedge (\mathsf{Ofc\_mem1}\ \mathrm{c}\ \mathrm{t}\ \Gamma) \wedge (\Delta = \cdot)\big) \\ \vee\ \big(\exists\ \mathrm{c}\ \mathrm{t}.\ (\mathrm{A} = \mathsf{OfcL}\ \mathrm{c}\ \mathrm{t}) \wedge (\Delta = [\,\mathrm{A}\,])\big) \\ \vee\ \big(\exists\ \mathrm{B}.\ (\mathrm{A} \leftarrow \mathrm{B}) \wedge (\Gamma,\,\Delta \vdash \mathrm{B})\big)\end{array}$$

The lemma SL_atom_ctxt1_inv inverts the derivation of an atom A under the invariant ctxt_invar1. Either it was derived by axJ, in which case it must be an Ofc atom; it was derived by axL, in which case it must be an OfcL atom; or it was derived by bc using one of the rules for OL judgments specified by prog ($\leftarrow$). (In Isabelle/HOL, it is actually stated in long goal format using **obtains**, and represented internally as an elimination rule.)

This is much more informative than what we could deduce without a context invariant, since the SL allows *any* atom to be taken from the context, which would defeat any attempt to deduce required premises for an OL rule.

There are three other lemmas of this kind, differing in which of the two context invariants is assumed and in whether the linear context is included in the SL statement to be inverted.

Lemma 7.20 (SL_atom_ctxt_inv)

$$\begin{array}{l}[\![\,\Gamma,\,\Delta \vdash \langle \mathrm{A}\rangle;\ \mathsf{ctxt\_invar}\ \Gamma\ \Delta\,]\!] \Longrightarrow \\ \quad \big(\exists\ \mathrm{c}\ \mathrm{t}.\ (\mathrm{A} = \mathsf{OfcL}\ \mathrm{c}\ \mathrm{t}) \wedge (\mathsf{Ofc\_mem1}\ \mathrm{c}\ \mathrm{t}\ \Gamma) \wedge (\Delta = [\,\mathrm{A}\,])\big) \\ \vee\ \big(\exists\ \mathrm{B}.\ (\mathrm{A} \leftarrow \mathrm{B}) \wedge (\Gamma,\,\Delta \vdash \mathrm{B})\big)\end{array}$$

Lemma 7.21 (SL_atomJ_ctxt1_inv)

$$\begin{array}{l}[\![\,\Gamma \vdash \langle \mathrm{A}\rangle;\ \mathsf{ctxt\_invar1}\ \Gamma\ \Delta\,]\!] \Longrightarrow \\ \quad \big(\exists\ \mathrm{c}\ \mathrm{t}.\ (\mathrm{A} = \mathsf{Ofc}\ \mathrm{c}\ \mathrm{t}) \wedge (\mathsf{Ofc\_mem1}\ \mathrm{c}\ \mathrm{t}\ \Gamma)\big) \\ \vee\ \big(\exists\ \mathrm{B}.\ (\mathrm{A} \leftarrow \mathrm{B}) \wedge (\Gamma \vdash \mathrm{B})\big)\end{array}$$

Lemma 7.22 (SL_atomJ_ctxt_inv)

$$\begin{array}{l}[\![\,\Gamma \vdash \langle \mathrm{A}\rangle;\ \mathsf{ctxt\_invar}\ \Gamma\ \Delta\,]\!] \Longrightarrow \\ \quad \big(\exists\ \mathrm{c}\ \mathrm{t}.\ (\mathrm{A} = \mathsf{Ofc}\ \mathrm{c}\ \mathrm{t}) \wedge (\mathsf{Ofc\_mem1}\ \mathrm{c}\ \mathrm{t}\ \Gamma) \wedge (\mathsf{OfcL\_mem1}\ \mathrm{c}\ \mathrm{t}\ \Delta)\big) \\ \vee\ \big(\exists\ \mathrm{B}.\ (\mathrm{A} \leftarrow \mathrm{B}) \wedge (\Gamma \vdash \mathrm{B})\big)\end{array}$$

Note that in the case of SL_atomJ_ctxt1_inv and SL_atomJ_ctxt_inv, the linear context Δ does not actually appear in the given SL statement. However, the context invariants defined for Γ and Δ together are still usable for such cases, and defining additional variants without Δ was considered an unnecessary complication.

There are a few more properties that will be needed to prove some of the semantic lemmas: the fact that ctxt_invar is stronger than ctxt_invar1 (ctxt_invar_invar1); downward closure of ctxt_invar1 for the linear context (ctxt_invar1_dc); and the existence of an OfcL atom corresponding to each Ofc atom under the stronger invariant (ctxt_invar_trans).

Lemma 7.23

ctxt_invar_invar1: ctxt_invar Γ Δ $\Longrightarrow$ ctxt_invar1 Γ Δ
ctxt_invar1_dc: $[\![$ ctxt_invar1 Γ Δ; mset Δ' $\sqsubseteq$ mset Δ $]\!]$ $\Longrightarrow$ ctxt_invar1 Γ Δ'
ctxt_invar_trans: $[\![$ ctxt_invar Γ Δ; Ofc c t $\in$ set Γ $]\!]$ $\Longrightarrow$ OfcL c t $\in$ mset Δ

It is essential to be able to build up the context invariant recursively starting from an empty context. This could be done by unfolding the definitions, but it is much more convenient to provide introduction rules, ctxt_invar_Nil_I and ctxt_invar_Cons_I:

Lemma 7.24

ctxt_invar_Nil_I: ctxt_invar $\cdot$ $\cdot$
ctxt_invar_Cons_I:
$[\![$ ctxt_invar Γ Δ; Ofc_fresh c Γ; OfcL_fresh c Δ $]\!]$
$\Longrightarrow$ ctxt_invar (Ofc c t # Γ) (OfcL c t # Δ)

7.4 Semantic lemmas and subject reduction

Lemma 7.25 (ofe_isterm)

$[\![\Gamma \vdash \langle$Ofe e t$\rangle$; ctxt_invar1 Γ Δ $]\!] \Longrightarrow \Gamma \vdash \langle$IsTerm e$\rangle$

The lemma ofe_isterm formalizes the fact that the typing judgment for expressions implies syntactic validity of its subject. It was proved using a variant of the

technique seen in Lemma 6.13, in which auxiliary functions are used to transform an arbitrary SL sequent, the transformation is shown to preserve SL provability by induction on SL_entails ($\vdash$), and then it is applied to the SL premise of the lemma to obtain the conclusion.

Definition 7.26

fun ofe_isterm_atmL :: *atm* $\Rightarrow$ *atm*
where ofe_isterm_atmL (Ofe e t) = IsTerm e
| ofe_isterm_atmL A = A

The function ofe_isterm_atmL is used to transform negative occurrences of SL atoms. It replaces (Ofe e t) with (IsTerm e) and leaves all other atoms untouched.

Definition 7.27

fun ofe_isterm_atmR :: *atm* $\Rightarrow$ *atm prp*
where ofe_isterm_atmR (IsTerm e) = $\langle$IsTerm e$\rangle$
| ofe_isterm_atmR (Ofe e t) = $\langle$IsTerm e$\rangle$
| ofe_isterm_atmR (Ofc c t) = $\langle$Ofc c t$\rangle$
| ofe_isterm_atmR A = $\top$

The function ofe_isterm_atmR is used to transform positive occurrences of SL atoms. It replaces (Ofe e t) with (IsTerm e), retains IsTerm and Ofc atoms, and replaces all other atoms with $\top$.

Definition 7.28

ofe_isterm_list $\equiv$ map ofe_isterm_atmL

The function ofe_isterm_list is used to transform each of the two context arguments of SL_entails, by applying ofe_isterm_atmL to each SL atom found there.

Definition 7.29

consts ofe_isterm_prp :: *atm prp* $\Rightarrow$ *atm prp*
primrec
ofe_isterm_prp $\langle$A$\rangle$ = ofe_isterm_atmR A

ofe_isterm_prp $\top = \top$
ofe_isterm_prp $\mathbf{1} = \mathbf{1}$
ofe_isterm_prp (B & C) = ofe_isterm_prp B & ofe_isterm_prp C
ofe_isterm_prp (B $\otimes$ C) = ofe_isterm_prp B $\otimes$ ofe_isterm_prp C
ofe_isterm_prp (A $\supset$ B) = ofe_isterm_atmL A $\supset$ ofe_isterm_prp B
ofe_isterm_prp (A $\multimap$ B) = ofe_isterm_atmL A $\multimap$ ofe_isterm_prp B
ofe_isterm_prp (forall F) = forall (λ x. ofe_isterm_prp (F x))

The function ofe_isterm_prp is used to transform the SL formula to be proved. It uses ofe_isterm_atmR for atoms coerced to (sub)formulas, and ofe_isterm_atmL for atoms on the left-hand side of implications.

Lemma 7.30 (ofe_isterm_seq)

Γ, $\Delta \vdash$ B $\Longrightarrow$ ofe_isterm_list Γ, ofe_isterm_list $\Delta \vdash$ ofe_isterm_prp B

The lemma ofe_isterm_seq is a strengthened form of ofe_isterm (Lemma 7.25) that is proved by induction on SL_entails. To obtain ofe_isterm as a corollary, the context-invariant premise is used to show that the context Γ is unchanged by ofe_isterm_list.

As in the previous formalizations, there are also several lemmas that are useful in the proof of subject reduction.

Lemma 7.31

ofsL_s_func:
$[\![\Gamma, \Delta \vdash \langle$OfsL s$\rangle$; ctxt_invar1 Γ Δ $]\!] \Longrightarrow$ s_func s
ofsL_s_lookup:
$[\![$ s_lookup s c v; ctxt_invar1 Γ Δ;
Γ, $\Delta \vdash \langle$OfsL s$\rangle$; $\Gamma \vdash \langle$Ofc c t$\rangle$ $]\!] \Longrightarrow \Gamma \vdash \langle$Ofe v t$\rangle$
ofsL_s_assign:
$[\![$ s_assign s c v s$'$; ctxt_invar1 Γ Δ;
Γ, $\Delta \vdash \langle$OfsL s$\rangle$; OfcL c t $\in$ mset Δ; $\Gamma \vdash \langle$Ofe v t$\rangle$ $]\!] \Longrightarrow \Gamma$, $\Delta \vdash \langle$OfsL s$'\rangle$
OfsL_s_fresh:
$[\![$ OfcL_fresh c Δ; ctxt_invar1 Γ Δ; Γ, $\Delta \vdash \langle$OfsL s$\rangle$ $]\!] \Longrightarrow$ s_fresh c s

The lemmas ofsL_s_func and ofsL_s_fresh correspond to ofs_s_func and (the reverse direction of) ofs_fresh_equiv, respectively, from Lemma 5.23. The lemmas ofsL_s_lookup and ofsL_s_assign correspond to Lemma 6.14 (s_lookup_ofs) and Lemma 6.15 (s_assign_ofs) respectively. Their statements differ from the previous versions in essentially the same ways noted for Definition 7.9.

These lemmas were proved by induction on the size of the Hybrid term s :: *state*, together with inversion of SL statements under the context invariant via Lemmas 7.19–7.22. This approach involved relatively long Isar proofs, including some multiset reasoning that was not well automated by Isabelle/HOL, as well as application of the induction hypothesis which tends to be difficult to automate for size induction.

For ofsL_s_func and OfsL_s_fresh, the only real alternative would be induction on SL sequents. Using the structural-induction rule provided by Isabelle/HOL for Definition 7.2, that would mean treating *every* SL rule as an induction step, not just the backchaining steps that correspond to OL rules. This would be further complicated by the need to strengthen each lemma to a property of arbitrary sequents (in the manner of Lemma 7.30) for the purpose of induction.

An alternative approach will be seen in Chapter 8, in which the SL is modified to support a form of size induction for SL sequents. This will produce proofs of similar structure and length to those by size induction on terms. Its advantage is applicability in cases where induction on terms cannot be used, and some form of induction on SL sequents is unavoidable (notably subject reduction in Lemma 8.20).

For ofsL_s_lookup and ofsL_s_assign, there is another option, namely induction on the predicates s_lookup and s_assign respectively. This would be a reasonable alternative and perhaps easier than size induction on s, but it was not tried.

Theorem 7.32 (Subject reduction)

$[\![\mathsf{eval}\ \mathrm{s}\ \mathrm{k}\ \mathrm{e}\ \mathrm{w};\ \mathsf{ctxt\_invar}\ \Gamma\ \Delta;$
$\Gamma,\ \Delta \vdash \langle \mathsf{OfsL}\ \mathrm{s} \rangle;\ \Gamma \vdash \langle \mathsf{Ofe}\ \mathrm{e}\ \mathrm{t} \rangle;\ \Gamma \vdash \langle \mathsf{Ofk}\ \mathrm{k}\ \mathrm{t}\ \mathrm{r} \rangle\ ]\!] \Longrightarrow \Gamma,\ \Delta \vdash \langle \mathsf{OffL}\ \mathrm{w}\ \mathrm{r} \rangle$

Proof. By induction on eval and cont (Definition 5.19), as in Theorem 6.16, together with a corresponding property for cont:

$$[\![\, \mathsf{cont}\ \mathrm{s}\ \mathrm{k}\ \mathrm{v}\ \mathrm{w};\ \mathsf{ctxt\_invar}\ \Gamma\ \Delta;\ \Gamma,\ \Delta \vdash \langle \mathsf{OfsL}\ \mathrm{s}\rangle;\ \Gamma \vdash \langle \mathsf{Ofe}\ \mathrm{v}\ \mathrm{t}\rangle;\ \Gamma \vdash \langle \mathsf{Ofk}\ \mathrm{k}\ \mathrm{t}\ \mathrm{r}\rangle \,]\!] \Longrightarrow \Gamma,\ \Delta \vdash \langle \mathsf{OffL}\ \mathrm{w}\ \mathrm{r}\rangle.$$

Case op_Ref_I**:** In this case, k is of the form (kRef k′) where k′ :: *cont*, w is of the form (fNew W) where W :: *cell* ⇒ *final* and (abstr W), and our induction hypothesis is

$$\bigwedge \mathrm{c}\ \Gamma\ \Delta\ \mathrm{t}.\ [\![\, \mathsf{s\_fresh}\ \mathrm{c}\ \mathrm{s};\ \mathsf{ctxt\_invar}\ \Gamma\ \Delta;\ \Gamma,\ \Delta \vdash \langle \mathsf{OfsL}\ (\mathsf{sCons}\ \mathrm{c}\ \mathrm{v}\ \mathrm{s})\rangle;\ \Gamma \vdash \langle \mathsf{Ofe}\ (\mathsf{eCell}\ \mathrm{c})\ \mathrm{t}\rangle;\ \Gamma \vdash \langle \mathsf{Ofk}\ \mathrm{k}'\ \mathrm{t}\ \mathrm{r}\rangle \,]\!] \Longrightarrow \Gamma,\ \Delta \vdash \langle \mathsf{OffL}\ (\mathrm{W}\ \mathrm{c})\ \mathrm{r}\rangle.$$

Let c be an arbitrary location. If it does not satisfy (Ofc_fresh c Γ), then by Definition 7.17, we must have (Ofc c t′) ∈ Γ for some t′ :: *tp*. Then by the rule offNewL_I1 from Definition 7.13, we have $\Gamma,\ \Delta \vdash \langle \mathsf{OffNewL}\ \mathrm{c}\ (\mathrm{W}\ \mathrm{c})\ \mathrm{r}\rangle$.

Otherwise, we must have (Ofc_fresh c Γ), and together with (ctxt_invar Γ Δ), that implies (OfcL_fresh c Δ) by unfolding the definitions. Together with the OfsL premise, it also implies (s_fresh c s) using ofsL_s_fresh from Lemma 7.31.

Let $\Gamma' = (\mathsf{Ofc}\ \mathrm{c}\ \mathrm{t}) \#\ \Gamma$ and $\Delta' = (\mathsf{OfcL}\ \mathrm{c}\ \mathrm{t}) \#\ \Delta$. We extend the context invariant to (ctxt_invar Γ′ Δ′) using Lemma 7.24 and the freshness conditions for c above.

From the Ofe premise, we deduce $\Gamma,\ \Delta \vdash \mathsf{J}\ \langle \mathsf{Ofe}\ \mathrm{v}\ \mathrm{t}\rangle$ by Lemma 7.7, and with the OfsL premise, we deduce

$$\Gamma,\ (\mathsf{OfcL}\ \mathrm{c}\ \mathrm{t}) \#\ \Delta \vdash \langle \mathsf{OfcL}\ \mathrm{c}\ \mathrm{t}\rangle \otimes (\mathsf{J}\ \langle \mathsf{Ofe}\ \mathrm{v}\ \mathrm{t}\rangle\ \&\ \langle \mathsf{OfsL}\ \mathrm{s}\rangle)$$

by the rules conjA_i, conjM_i, and axL from Definition 7.2. We then use the rule ofsL_Cons_I from Definition 7.12, together with bc and the definition of Δ′, to deduce $\Gamma,\ \Delta' \vdash \langle \mathsf{OfsL}\ (\mathsf{sCons}\ \mathrm{c}\ \mathrm{v}\ \mathrm{s})\rangle$.

We have $\Gamma' \vdash \langle \mathsf{Ofe}\ (\mathsf{eCell}\ \mathrm{c})\ (\mathsf{tRef}\ \mathrm{t})\rangle$ by the rules bc, ofe_Cell_I, and axJ, since $(\mathsf{Ofc}\ \mathrm{c}\ \mathrm{t}) \in \mathsf{set}\ \Gamma'$ by construction.

Using Lemma 7.22, the Ofk premise must have been deduced using bc, and inverting Definition 7.11, the rule used must have been ofk_Ref_I. Thus we have $\Gamma \vdash \langle \mathsf{Ofk}\ \mathrm{k}'\ (\mathsf{tRef}\ \mathrm{t})\ \mathrm{r}\rangle$.

We now have all the premises needed to apply the induction hypothesis with $\mathrm{c} \mapsto \mathrm{c}$, $\Gamma \mapsto \Gamma'$, $\Delta \mapsto \Delta'$, and $\mathrm{t} \mapsto (\mathsf{tRef}\ \mathrm{t})$; for two of them, Γ is weakened to Γ' using Lemma 7.3. Unfolding the definitions of Γ' and Δ', the conclusion is

$$(\mathsf{Ofc}\ \mathrm{c}\ \mathrm{t}) \#\ \Gamma,\ (\mathsf{OfcL}\ \mathrm{c}\ \mathrm{t}) \#\ \Delta \vdash \langle \mathsf{OffL}\ (\mathrm{W}\ \mathrm{c})\ \mathrm{r}\rangle$$

from which we deduce $\Gamma, \Delta \vdash \langle \mathsf{OffNewL}\ \mathrm{c}\ (\mathrm{W}\ \mathrm{c})\ \mathrm{r}\rangle$ by the rules impL_i and impJ_i from Definition 7.2 and offNewL_I from Definition 7.13.

That is the same statement that we deduced in the case where $(\mathsf{Ofc\_fresh}\ \mathrm{c}\ \Gamma)$ is not true, so it must hold for arbitrary c without any assumption of freshness or non-freshness for Γ. By the rule all_i, we deduce

$$\Gamma, \Delta \vdash \mathsf{all}\ \mathrm{c}.\ \langle \mathsf{OffNewL}\ \mathrm{c}\ (\mathrm{W}\ \mathrm{c})\ \mathrm{r}\rangle,$$

and together with $(\mathsf{abstr}\ \mathrm{W})$, we apply the rules bc and offL_New_I to reach the required conclusion $\Gamma, \Delta \vdash \langle \mathsf{OffL}\ (\mathsf{fNew}\ \mathrm{W})\ \mathrm{r}\rangle$.

The remaining cases were covered by a single automatic proof method, except for op_Deref_I and op_Assign2_I which used the lemmas ofsL_s_lookup and ofsL_s_assign respectively from Lemma 7.31. □

Chapter 8

Case Study: Evaluation in the SL

The third two-level formalization uses the linear SL of the previous formalization, but represents evaluation in the SL along with typing, rather than reusing the meta-level evaluation predicates shared by the previous three formalizations. This change was intended to further exploit the SL's linear context as a way to represent the contents of memory during evaluation, and as an intermediate step toward the use of an ordered SL to represent continuations, which will be seen in Chapter 9.

It also led to changes in the SL. The subject reduction theorem and several of its key lemmas have evaluation statements as premises, and representing these statements in the unmodified SL from Chapter 7 would provide only awkward forms of induction. Building on previous work [46] that uses a natural-number argument in SL statements as an induction measure, the SL is augmented with an *ordinal* argument to permit (transfinite) induction on the height of a derivation. The use of ordinals rather than natural numbers allows the entire SL from Chapter 7 – including its infinitely-branching all_i rule – to be derived by existentially quantifying the derivation-height argument.

This formalization was constructed by modifying the previous two-level linear formalization from Chapter 7. The syntax used for the previous three formalizations

and described in Section 5.1 (excluding the definition of typing contexts) is retained here, along with the auxiliary predicate s_fresh from Definition 5.9. The SL representation of typing judgments from Section 7.2 is also reused without modification. The meta-level evaluation predicates from Section 5.3, on the other hand, will be replaced by SL atoms in Section 8.2.

8.1 Specification logic

The type *prp* of SL formulas is unchanged from Definition 7.1, and is restated here in abbreviated form:

$$\textbf{datatype}\ a\ prp = \langle \mathrm{A} \rangle \mid \top \mid \mathbf{1} \mid \mathrm{B}\ \&\ \mathrm{C} \mid \mathrm{B} \otimes \mathrm{C} \mid \mathrm{A} \supset \mathrm{B} \mid \mathrm{A} \multimap \mathrm{B} \mid \mathsf{all}\ \mathrm{x}.\ \mathrm{B}$$

where A :: *a* and B, C :: *a prp*.

The predicate SL_entails from Definition 7.2, used to express SL statements, is extended with an argument h :: *ordinal* that functions as a bound on derivation height for the purpose of induction. (As in the previous chapter, the linear context may be omitted when empty.) Lemma names are prefixed with "hSL" rather than "SL", and rule names primed, to distinguish them from the versions without the ordinal argument that will be derived later.

Definition 8.1 (SL sequent rules)

class ATM = TYPE + **fixes** prog :: $[\, a,\ a\ prp \,] \Rightarrow bool$ $\quad$ (*notation* A $\leftarrow$ B)
inductive hSL_entails :: $[\, ordinal,\ (a :: \mathrm{ATM})\ list,\ a\ list,\ a\ prp \,] \Rightarrow bool$
(*notation* h, Γ, $\Delta \Vdash$ B)
where bc′: h, Γ, $\Delta \Vdash \langle \mathrm{A} \rangle \Longleftarrow [\![\, \mathrm{A} \leftarrow \mathrm{B};\ \mathrm{h}' < \mathrm{h};\ \mathrm{h}', \Gamma, \Delta \Vdash \mathrm{B} \,]\!]$
| axJ′: h, Γ, $\cdot \Vdash \langle \mathrm{A} \rangle \Longleftarrow \mathrm{A} \in \mathsf{set}\ \Gamma$
| axL′: h, Γ, $[\,\mathrm{A}\,] \Vdash \langle \mathrm{A} \rangle$
| ttA_i′: h, Γ, $\Delta \Vdash \top$
| ttM_i′: h, Γ, $\cdot \Vdash \mathbf{1}$
| conjA_i′: h, Γ, $\Delta \Vdash \mathrm{B}\ \&\ \mathrm{C} \Longleftarrow [\![\, \mathrm{h}, \Gamma, \Delta \Vdash \mathrm{B};\ \mathrm{h}, \Gamma, \Delta \Vdash \mathrm{C} \,]\!]$

$\mid$ conjM_i′: h, Γ, Δ ⊩ B ⊗ C ⟸ ⟦ h, Γ, Δ_L ⊩ B; h, Γ, Δ_R ⊩ C;
mset Δ_L + mset Δ_R = mset Δ ⟧
$\mid$ impJ_i′: h, Γ, Δ ⊩ A ⊃ B ⟸ h, A # Γ, Δ ⊩ B
$\mid$ impL_i′: h, Γ, Δ ⊩ A ⊸ B ⟸ h, Γ, A # Δ ⊩ B
$\mid$ all_i′: h, Γ, Δ ⊩ all x. B x ⟸ (⋀ x. h, Γ, Δ ⊩ B x)

The ordinal argument h passes unchanged through all the rules except bc′. This will be convenient in proofs by induction, because backchaining steps typically correspond to OL rules. (A height measure that counts every SL rule could be reconstructed as the lexical combination of h and the height of the formula to be proved.)

The rule bc′ is written with the heights of both premise and conclusion as variables, related by an inequality $h' < h$. With natural-number derivation heights, we could simply increment the height in passing from premise to conclusion; but since there are *limit ordinals* (such as ω) that are not successors of any other ordinal, that form of the rule would be weaker here and would not admit straightforward height-weakening. (The inequality form also gives cleaner induction proofs.)

Lemma 8.2

hSL_structural:
⟦ h, Γ, Δ ⊩ B; h ⩽ h′; set Γ ⊆ set Γ′; mset Δ = mset Δ′ ⟧ ⟹ h′, Γ′, Δ′ ⊩ B

With the additional argument h comes a new structural rule: height-weakening, which allows the ordinal argument of an SL statement to be replaced with any larger ordinal. This is combined with the other structural rules from Lemma 7.3 in the lemma hSL_structural.

Lemma 8.3

hSL_cutJ: ⟦ h_B, A # Γ, Δ ⊩ B; h_A, Γ ⊩ ⟨A⟩ ⟧ ⟹ $h_A + h_B$, Γ, Δ ⊩ B
hSL_cutL: ⟦ h_B, Γ, A # Δ_1 ⊩ B; h_A, Γ, Δ_2 ⊩ ⟨A⟩;
mset Δ = mset Δ_1 + mset Δ_2 ⟧ ⟹ $h_A + h_B$, Γ, Δ ⊩ B

For the derived cut rules, it is necessary to calculate a derivation-height bound for the conclusion from similar bounds for the premises. The obvious approach is to add the bounds, but ordinal addition is not commutative, so there is a choice between $h_A + h_B$ and $h_B + h_A$. Since the proof requires cancellation of h_A, and ordinal sums admit cancellation only on the left, only the former choice succeeds.

In more general terms, for lemmas like these that piece together a derivation of the conclusion from derivations of the premises, the calculated bound must be strictly monotonic with respect to the premise that contributes to the root of the derivation, but need only be monotonic with respect to any other premises. This fits nicely with the properties of ordinal addition, which is strictly monotonic in its right operand but only monotonic in its left operand.

Definition 7.6 (J) is reused without modification, while Lemmas 7.5 (derived elimination rules) and 7.7 (simplification for J) are reused with the addition of an h argument to each SL statement.

Definition 8.4

$A \circledcirc B \equiv \langle A \rangle \otimes B$

The derived connective $\circledcirc$ (called LCONS in the formal theory) is used operationally as the reverse of $\multimap$: it removes the atom A from the linear context. This atom should not have any backchaining rules, nor should it occur in the intuitionistic context. Intro and elim rules are provided that demand these conditions as premises, and are marked as "safe" so they can be applied without backtracking; as such, this connective improves proof automation when used for its intended purpose, but would lead proof search to a dead end if the conditions were not met.

Definition 8.5 (SL statements without h)

SL_entails :: $\big[(a :: \text{ATM})\ list,\ a\ list,\ a\ prp\big] \Rightarrow bool$ (*notation* $\Gamma, \Delta \vdash B$)
where $(\Gamma, \Delta \vdash B) = (\exists\, h.\ (h, \Gamma, \Delta \Vdash B))$

With this definition, all the rules of Definition 7.2 are derivable from Definition 8.1. Most are straightforward; bc uses the successor function on ordinals. The one interesting case is all_i:

Lemma 8.6

all_i: $\Gamma, \Delta \vdash \mathsf{all}\ x.\ B\ x \Longleftarrow (\bigwedge x.\ \Gamma, \Delta \vdash B\ x)$

Proof. By Definition 8.5, the premise is equivalent to

$$\forall x.\ \exists h.\ (h, \Gamma, \Delta \Vdash B\ x).$$

By the Axiom of Choice, h may be represented as a function H :: *exp* ⇒ *ordinal* applied to x:

$$\exists H.\ \forall x.\ (H\ x, \Gamma, \Delta \Vdash B\ x).$$

Let $h_0 = (\mathsf{oSup}\ H)$; by definition, we have $(H\ x) \leqslant h_0$ for all x :: *exp*. By height-weakening (Lemma 8.2), we have $\forall x.\ (h_0, \Gamma, \Delta \Vdash B\ x)$. Now we may apply the rule all_i′ to obtain $h_0, \Gamma, \Delta \Vdash \mathsf{all}\ x.\ B\ x$, and using Definition 8.5 again, conclude

$$\Gamma, \Delta \vdash \mathsf{all}\ x.\ B\ x$$

as was to be proven. □

This proof makes essential use of the Axiom of Choice, so this kind of SL would not work in a constructive setting. It also uses the supremum function

$$\mathsf{oSup} :: ((x :: \textsc{countable}) \Rightarrow \mathit{ordinal}) \Rightarrow \mathit{ordinal},$$

which the Ordinal theory provides only for functions on *countable* domains[1], so it is necessary to prove that *exp* (Definition 5.2) is countable. This is done compositionally by proving that *con* is countable and that (*a expr*) is countable for any *a* :: COUNTABLE, which is straightforward but tedious, as the proofs involve actually "counting" the constructors (i.e., mapping them to distinct natural numbers).

[1] Actually the Ordinal theory only provides a function oLimit :: (*nat* ⇒ *ordinal*) ⇒ *ordinal*; we define oSup in terms of oLimit.

It would be possible to also derive a structural induction rule for SL_entails, and thus make it fully equivalent to the predicate of the same name in Section 7.1. But this was considered unnecessary, since the purpose of the ordinal argument was to provide a more convenient alternative to structural induction. However, a case-analysis lemma was proved to allow inversion of the SL rules without unfolding Definition 8.5.

8.2 Evaluation judgments

The type of atomic formulas has 13 constructors, of which the first five are used to represent evaluation and the other eight to represent typing. The latter are unchanged from Definition 7.8.

Definition 8.7

datatype *atm* = EvalL *cont exp final* | ContL *cont exp final*
| OpRefL *cell cont exp final* | PackStateL *state* | CellL *cell exp*
| IsTerm *exp* | Ofe *exp tp* | Ofk *cont tp tp* | Ofc *cell tp*
| OfcL *cell tp* | OfsL *state* | OffL *final tp* | OffNewL *cell final tp*

The evaluation judgments (EvalL, ContL, OpRefL, and PackStateL) use their own SL context, completely separate from that used for typing. It consists of CellL atoms in the linear context, assigning values to distinct locations. This context will be related to the typing context by the context invariant, or sometimes less directly via shared terms in separate SL statements (e.g., the *state* arguments of PackStateL and OfsL in Theorem 8.23).

The SL context for evaluation corresponds to the *state* arguments of eval and cont in Definition 5.19. However, it is tricky to use the SL context for both the inputs and outputs of evaluation. For this reason, the mapping of locations to values is transformed by PackStateL into a *state* for output.

The specification of the typing judgments is unchanged from the previous formalization (Definitions 7.9 to 7.13). The SL predicates EvalL and ContL correspond

to the meta-level eval and cont from Definition 5.19. OpRefL is an auxiliary predicate used by ContL to relativize a universal quantification over locations, much like OffNewL as used by OffL. PackStateL relates the contents of memory as represented in the SL context with a state in the sense of Definition 5.6.

Definition 8.8 (Specification of SL atoms, part 1 – EvalL, ContL, OpRefL)

inductive prog_atm :: $\left[\, atm,\ atm\ prp \,\right] \Rightarrow bool$ $\quad$ ($notation$ A ← B)

where ev_Zero_I: EvalL k eZero w ← ⟨ContL k eZero w⟩

| ev_Suc_I: EvalL k (eSuc e) w ← ⟨EvalL (kCons (λ v. eSuc v) k) e w⟩

| ev_Pred_I: EvalL k (ePred e) w ← ⟨EvalL (kPred k) e w⟩

| ev_IsZero_I: EvalL k (eIsZero e) w ← ⟨EvalL (kIsZero k) e w⟩

| ev_True_I: EvalL k eTrue w ← ⟨ContL k eTrue w⟩

| ev_False_I: EvalL k eFalse w ← ⟨ContL k eFalse w⟩

| ev_IfThen_I: EvalL k (eIfThen c tt ff) w ← ⟨EvalL (kIfThen tt ff k) c w⟩

| ev_Unit_I: EvalL k eUnit w ← ⟨ContL k eUnit w⟩

| ev_Pair_I: EvalL k (ePair x y) w ← ⟨EvalL (kPair1 y k) x w⟩

| ev_Fst_I: EvalL k (eFst p) w ← ⟨EvalL (kFst k) p w⟩

| ev_Snd_I: EvalL k (eSnd p) w ← ⟨EvalL (kSnd k) p w⟩

| ev_App_I: EvalL k (f $ x) w ← ⟨EvalL (kArg x k) f w⟩

| ev_Fn_I: EvalL k (eFn F) w ← ⟨ContL k (eFn F) w⟩ ⟸ abstr F

| ev_Fix_I: EvalL k (eFix F) w ← ⟨EvalL k (F (eFix F)) w⟩ ⟸ abstr F

| ev_Letv_I: EvalL k (eLetv v F) w ← ⟨EvalL k (F v) w⟩ ⟸ abstr F

| ev_Ref_I: EvalL k (eRef e) w ← ⟨EvalL (kRef k) e w⟩

| ev_Deref_I: EvalL k (eDeref e) w ← ⟨EvalL (kDeref k) e w⟩

| ev_Assign_I: EvalL k (eAssign m e) w ← ⟨EvalL (kAssign1 e k) m w⟩

| ev_Cell_I: EvalL k (eCell c) w ← ⟨ContL k (eCell c) w⟩

| op_Cons_I: ContL (kCons E k) v w ← ⟨ContL k (E v) w⟩ ⟸ abstr E

| op_Arg_I: ContL (kArg x k) (eFn F) w ← ⟨EvalL (kApp F k) x w⟩ ⟸ abstr F

| op_App_I: ContL (kApp E k) v w ← ⟨EvalL k (E v) w⟩ ⟸ abstr E

| op_Pred_I1: ContL (kPred k) eZero w ← ⟨ContL k eZero w⟩

| op_Pred_I2: ContL (kPred k) (eSuc v) w ← ⟨ContL k v w⟩

| op_IsZero_I1: ContL (kIsZero k) eZero w ← ⟨ContL k eTrue w⟩

| op_IsZero_I2: ContL (kIsZero k) (eSuc v) w ← ⟨ContL k eFalse w⟩

| op_IfThen_I1: ContL (kIfThen tt ff k) eTrue w ← ⟨EvalL k tt w⟩

$$
\begin{array}{l}
\mid \text{op\_IfThen\_I2}\colon \mathsf{ContL}\ (\mathsf{kIfThen}\ \mathrm{tt}\ \mathrm{ff}\ \mathrm{k})\ \mathsf{eFalse}\ \mathrm{w} \leftarrow \langle \mathsf{EvalL}\ \mathrm{k}\ \mathrm{ff}\ \mathrm{w}\rangle \\
\mid \text{op\_Pair1\_I}\colon \mathsf{ContL}\ (\mathsf{kPair1}\ \mathrm{y}\ \mathrm{k})\ \mathrm{v}\ \mathrm{w} \leftarrow \langle \mathsf{EvalL}\ (\mathsf{kPair2}\ \mathrm{v}\ \mathrm{k})\ \mathrm{y}\ \mathrm{w}\rangle \\
\mid \text{op\_Pair2\_I}\colon \mathsf{ContL}\ (\mathsf{kPair2}\ \mathrm{v}\ \mathrm{k})\ \mathrm{v}'\ \mathrm{w} \leftarrow \langle \mathsf{ContL}\ \mathrm{k}\ (\mathsf{ePair}\ \mathrm{v}\ \mathrm{v}')\ \mathrm{w}\rangle \\
\mid \text{op\_Fst\_I}\colon \mathsf{ContL}\ (\mathsf{kFst}\ \mathrm{k})\ (\mathsf{ePair}\ \mathrm{v}\ \mathrm{v}')\ \mathrm{w} \leftarrow \langle \mathsf{ContL}\ \mathrm{k}\ \mathrm{v}\ \mathrm{w}\rangle \\
\mid \text{op\_Snd\_I}\colon \mathsf{ContL}\ (\mathsf{kSnd}\ \mathrm{k})\ (\mathsf{ePair}\ \mathrm{v}\ \mathrm{v}')\ \mathrm{w} \leftarrow \langle \mathsf{ContL}\ \mathrm{k}\ \mathrm{v}'\ \mathrm{w}\rangle \\
\mid \text{op\_Ref\_I}\colon \mathsf{ContL}\ (\mathsf{kRef}\ \mathrm{k})\ \mathrm{v}\ (\mathsf{fNew}\ \mathrm{W}) \\
\qquad\qquad \leftarrow \mathsf{all}\ \mathrm{c}.\ \langle \mathsf{OpRefL}\ \mathrm{c}\ \mathrm{k}\ \mathrm{v}\ (\mathrm{W}\ \mathrm{c})\rangle \Longleftarrow \mathsf{abstr}\ \mathrm{W} \\
\mid \text{op\_Deref\_I}\colon \mathsf{ContL}\ (\mathsf{kDeref}\ \mathrm{k})\ (\mathsf{eCell}\ \mathrm{c})\ \mathrm{w} \\
\qquad\qquad \leftarrow (\mathsf{CellL}\ \mathrm{c}\ \mathrm{v} \otimes \top)\ \&\ \langle \mathsf{ContL}\ \mathrm{k}\ \mathrm{v}\ \mathrm{w}\rangle \\
\mid \text{op\_Assign1\_I}\colon \mathsf{ContL}\ (\mathsf{kAssign1}\ \mathrm{e}\ \mathrm{k})\ (\mathsf{eCell}\ \mathrm{c})\ \mathrm{w} \leftarrow \langle \mathsf{EvalL}\ (\mathsf{kAssign2}\ \mathrm{c}\ \mathrm{k})\ \mathrm{e}\ \mathrm{w}\rangle \\
\mid \text{op\_Assign2\_I}\colon \mathsf{ContL}\ (\mathsf{kAssign2}\ \mathrm{c}\ \mathrm{k})\ \mathrm{v}\ \mathrm{w} \\
\qquad\qquad \leftarrow \mathsf{CellL}\ \mathrm{c}\ \mathrm{v}' \otimes (\mathsf{CellL}\ \mathrm{c}\ \mathrm{v} \multimap \langle \mathsf{ContL}\ \mathrm{k}\ \mathrm{v}\ \mathrm{w}\rangle) \\
\mid \text{op\_Done\_I}\colon \mathsf{ContL}\ \mathsf{kDone}\ \mathrm{v}\ (\mathsf{fVal}\ \mathrm{s}\ \mathrm{v}) \leftarrow \langle \mathsf{PackStateL}\ \mathrm{s}\rangle \\
\mid \text{opRefL\_I1}\colon \mathsf{OpRefL}\ \mathrm{c}\ \mathrm{k}\ \mathrm{v}\ \mathrm{w} \leftarrow \mathsf{CellL}\ \mathrm{c}\ \mathrm{v}' \otimes \top \\
\mid \text{opRefL\_I}\colon \mathsf{OpRefL}\ \mathrm{c}\ \mathrm{k}\ \mathrm{v}\ \mathrm{w} \leftarrow \mathsf{CellL}\ \mathrm{c}\ \mathrm{v} \multimap \langle \mathsf{ContL}\ \mathrm{k}\ (\mathsf{eCell}\ \mathrm{c})\ \mathrm{w}\rangle
\end{array}
$$

Most of the evaluation rules correspond directly to those of Definition 5.19, with the state argument s removed since that information has been moved to the SL context. There were no connectives to be translated, as continuation-style operational semantics is non-branching; the work of evaluation is done by pattern-matching and construction of terms, which are unaffected by the change from meta-level to specification-level representation.

The interesting cases are the rules that deal specifically with reference cells. op_Deref_I and op_Assign2_I use the multiplicative conjunction and linear implication to look up and replace values for locations given by CellL atoms in the linear context; in the case of op_Deref_I, the additive conjunction and truth constant are also used, so that the CellL atom is not removed from the context for the next evaluation step. op_Ref_I uses HOAS for the newly allocated location, and uses the same technique as offL_New_I (Definition 7.13) to quantify over fresh locations only.

The finishing-up rule op_Done_I is also interesting, as it needs to return the final state as a term s :: *state* in an answer of the form (fVal s v), but the contents of

memory are represented in the SL context during computation. The translation from SL context to *state* is performed by a separate SL predicate PackStateL.

Definition 8.9 (Specification of SL atoms, part 2 – PackStateL)

inductive prog_atm :: $[\, atm,\ atm\ prp \,] \Rightarrow bool$ (*notation* A ← B)
where ps_Nil_I: PackStateL sNil ← **1**
| ps_Cons_I: PackStateL (sCons c v s)
← CellL c v ⊗ ⟨PackStateL s⟩ ⟸ s_fresh c s

The rule ps_Cons_I uses the multiplicative conjunction to extract a CellL atom from the linear context, converting its location-value pair into an sCons constructor applied to a term of type *state*. The rule ps_Nil_I uses the multiplicative truth constant, which requires an empty linear context, to ensure that all such atoms are processed.

CellL atoms can be processed in any order, in contrast to Definition 5.19 which constructs the state in order of allocation. Since the resulting state is part of the answer (output), and its use of an association list to represent a function is an implementation detail, this difference is irrelevant.

8.3 Context invariants

As in Section 7.3, a context invariant is needed to reason about SL statements. It is complicated by the need to relate separate SL contexts for evaluation and typing. To deal with this complexity and improve proof automation, the context invariant was built in a modular way using Isabelle's locale mechanism.

Definition 8.10

locale ctxt_abstr =
fixes S :: *cell set* **and** V :: *cell* ⇒ *exp* **and** T :: *cell* ⇒ *tp*
assumes finite_S: finite S

This statement defines ctxt_abstr as a *locale*, i.e., a separate Isabelle context in which lemmas and definitions may be stated. Within the locale, S, V, and T are fixed parameters of the specified types (as if declared with **consts**), and finite_S is an axiom stating that S is finite. All the definitions and lemmas of the enclosing theory are also available, whether stated before or after the locale definition.

It also defines a predicate ctxt_abstr in the enclosing theory to represent the locale axioms:

ctxt_abstr S V T = finite S.

(In fact Isabelle 2008 optimizes away the arguments V and T that are not used in the locale axioms, but we will use the unoptimized form.)

When a lemma is stated "(**in** ctxt_abstr)", it is *exported* to the enclosing theory, by universally quantifying the locale parameters and introducing a premise (ctxt_abstr S V T). A proof of the lemma also proves its exported form, since the locale parameters may be generalized and the locale axioms may be replaced with use of the ctxt_abstr premise. The locale axioms themselves are also exported, e.g., finite_S:

ctxt_abstr.finite_s: ctxt_abstr S V T $\Longrightarrow$ finite S.

In accordance with Isabelle's definitional style of reasoning, this is a proven statement rather than an axiom in the enclosing theory, thanks to its ctxt_abstr premise.

Definitions may likewise be stated "(**in** ctxt_abstr)", and they are exported by turning any occurrences of the locale parameters into additional parameters of the definition. (The locale axioms are not involved, as Isabelle does not have dependent types and explicit proof terms.)

The use of locales helps to improve proof automation, since Isabelle's automatic proof methods can be set up with rules *within* the locale that depend on the locale parameters and axioms in ways that would not be supported for their exported forms. However, to simplify the presentation of the formalization, the lemmas below will be

given in exported form.

The locale ctxt_abstr defines an "abstract context", i.e., a straightforward Isabelle/HOL representation of the information represented in the SL evaluation and typing contexts. The parameter S represents the set of allocated locations; V is a function assigning values to locations (corresponding to CellL atoms in the evaluation context); and T is a function assigning types to locations (corresponding to Ofc and OfcL atoms in the typing context).

Definition 8.11

locale ctxt_D = ctxt_abstr +
  fixes Δ :: *atm list*
  assumes D_eltm:
    count (mset Δ) a =
      nat_of_bool(case a of CellL c v ⇒ c ∈ S ∧ v = V c | _ ⇒ False)

The locale ctxt_D relates the linear part of the SL evaluation context, Δ, to the abstract context defined by ctxt_abstr, stating that an SL atom a occurs in Δ iff it is of the form (CellL c v) where c ∈ S and v = (V c). (The function nat_of_bool maps True to 1 and False to 0.) The intuitionistic part of the SL evaluation context is empty, and this will be stated explicitly (as in Chapter 6) rather than via a context invariant.

Definition 8.12

locale ctxt_G′ = ctxt_abstr +
  fixes Γ′ :: *atm list*
  assumes G′_eltm:
    count (mset Γ′) a =
      nat_of_bool(case a of Ofc c t ⇒ c ∈ S ∧ t = T c | _ ⇒ False)
locale ctxt_D′ = ctxt_abstr +
  fixes Δ′ :: *atm list*
  assumes D′_eltm:
    count (mset Δ′) a =
      nat_of_bool(case a of OfcL c t ⇒ c ∈ S ∧ t = T c | _ ⇒ False)

The locale ctxt_G′ relates the intutionistic part of the SL typing context, Γ′, to the abstract context defined by ctxt_abstr, stating that an SL atom a occurs in Γ′ iff it is of the form (Ofc c t) where c ∈ S and t = (T c). The locale ctxt_D′ similarly relates the linear part of the SL typing context, Δ′, to the abstract context.

Definition 8.13

locale ctxt_invar = ctxt_D + ctxt_G′ + ctxt_D′ +
  assumes type_correct: c ∈ S ⟹ Γ′ ⊢ ⟨Ofe (V c) (T c)⟩

The locale ctxt_invar brings together the locales for the SL evaluation and typing contexts, and adds the condition that for each location c ∈ S, the value (V c) must have the type (T c) in the context Γ′. (That condition will stand in for the OfsL premise of Theorem 7.32, in the inductive proof of subject reduction.) It defines a predicate with six arguments, (ctxt_invar S V T Δ Γ′ Δ′).

Lemma 8.14

D_S_down:
  ⟦ ctxt_D S V T Δ; mset Δ = mset Δ_s + {{ CellL c v }} ⟧ ⟹
    ∃ S_s. S = S_s ∪ {c} ∧ c ∉ S_s ∧ v = V c ∧ ctxt_D S_s V T Δ_s
S_D′_down:
  ⟦ ctxt_D′ S V T Δ′; S = S_s ∪ {c}; c ∉ S_s ⟧ ⟹
    ∃ Δ'_s. mset Δ′ = mset Δ'_s + {{ OfcL c (T c) }} ∧ ctxt_D′ S_s V T Δ'_s

The lemma D_S_down allows a CellL atom to be removed from Δ, while maintaining the invariant ctxt_D by removing the corresponding location from S. The lemma S_D′_down similarly allows a location c to be removed from S, while maintaining the invariant ctxt_D′ by removing (OfcL c (T c)) from Δ′.

These lemmas will be used in reasoning about OfsL; by using ctxt_D and ctxt_D′ separately, it will not be necessary to define an analogue of ctxt_invar1 from Section 7.3. This is an advantage of the modular approach to specifying ctxt_invar.

Lemma 8.15

ctxt_invar_Cons_I:

$$[\![\, \mathsf{ctxt\_invar}\ \mathrm{S}\ \mathrm{V}\ \mathrm{T}\ \Delta\ \Gamma'\ \Delta';\ \Gamma' \vdash \langle \mathsf{Ofe}\ \mathrm{v}\ \mathrm{t}\rangle;\ \mathrm{c} \notin \mathrm{S}\,]\!] \Longrightarrow$$
$$\mathsf{ctxt\_invar}\ (\mathrm{S} \cup \{\mathrm{c}\})\ (\mathrm{V}\ (\mathrm{c} := \mathrm{v}))\ (\mathrm{T}\ (\mathrm{c} := \mathrm{t}))$$
$$(\mathsf{CellL}\ \mathrm{c}\ \mathrm{v}\ \#\ \Delta)\ (\mathsf{Ofc}\ \mathrm{c}\ \mathrm{t}\ \#\ \Gamma')\ (\mathsf{OfcL}\ \mathrm{c}\ \mathrm{t}\ \#\ \Delta')$$

The lemma ctxt_invar_Cons_I corresponds to the lemma of the same name in Lemma 7.24, and allows the context invariant to be extended with a new location. The use of ctxt_abstr allows freshness to be checked in the simpler form $\mathrm{c} \notin \mathrm{S}$. With evaluation now represented in the SL along with typing, it is necessary to specify both a value v and a type t for the new location c, and they are related by the Ofe premise to satisfy the locale axiom type_correct.

Lemma 8.16

structural:

$$[\![\, \mathsf{ctxt\_invar}\ \mathrm{S}\ \mathrm{V}\ \mathrm{T}\ \Delta\ \Gamma'\ \Delta';$$
$$\mathsf{mset}\ \Delta = \mathsf{mset}\ \Delta_\mathrm{p};\ \mathsf{mset}\ \Gamma' = \mathsf{mset}\ \Gamma'_\mathrm{p};\ \mathsf{mset}\ \Delta' = \mathsf{mset}\ \Delta'_\mathrm{p}\,]\!] \Longrightarrow$$
$$\mathsf{ctxt\_invar}\ \mathrm{S}\ \mathrm{V}\ \mathrm{T}\ \Delta_\mathrm{p}\ \Gamma'_\mathrm{p}\ \Delta'_\mathrm{p}$$

This lemma allows permutation of all of the SL-context arguments of ctxt_invar. It will be used to support reasoning about SL contexts as multisets rather than lists.

A set of inversion lemmas were proved for atomic SL statements in the contexts $(\cdot, \Delta)$, $(\Gamma', \cdot)$, and (Γ', Δ'). These are similar to Lemmas 7.19 to 7.22, and their statements are omitted here. They are provided for SL statements both with and without ordinal arguments.

Lemma 8.17

ctxt_value_subst:

$$[\![\, \mathsf{ctxt\_invar}\ \mathrm{S}\ \mathrm{V}\ \mathrm{T}\ (\mathsf{CellL}\ \mathrm{c}\ \mathrm{v}\ \#\ \Delta)\ \Gamma'\ \Delta';\ \mathsf{Ofc}\ \mathrm{c}\ \mathrm{t} \in \mathsf{set}\ \Gamma';\ \Gamma' \vdash \langle \mathsf{Ofe}\ \mathrm{v}'\ \mathrm{t}\rangle\,]\!] \Longrightarrow$$
$$\mathsf{ctxt\_invar}\ \mathrm{S}\ (\mathrm{V}\ (\mathrm{c} := \mathrm{v}'))\ \mathrm{T}\ (\mathsf{CellL}\ \mathrm{c}\ \mathrm{v}'\ \#\ \Delta)\ \Gamma'\ \Delta'$$

The lemma ctxt_value_subst allows the replacement of one value v with another value v′ for a location c in the context invariant, so long as the new value has the appropriate type. It is used to prove type-soundness of assignment.

8.4 Semantic lemmas and subject reduction

Lemma 8.18

PackStateL_OfsL:

$[\![\, \mathsf{ctxt\_invar}\ S\ V\ T\ \Delta\ \Gamma'\ \Delta';\ \cdot,\ \Delta \vdash \langle \mathsf{PackStateL}\ s \rangle \,]\!] \Longrightarrow \Gamma',\ \Delta' \vdash \langle \mathsf{OfsL}\ s \rangle$

The lemma PackStateL_OfsL carries the type information in the locale axiom type_correct across the translation from SL evaluation context to state performed by PackStateL, to conclude that the resulting state s satisfies OfsL.

It was proved by transfinite induction on the SL's ordinal argument, after unfolding Definition 8.5 in the PackStateL premise. This is reasonably straightforward, and results in essentially the same proof structure that would be obtained with ordinary induction on a natural-number derivation-height argument. However, as noted after Lemma 7.31, size induction is more difficult to automate than structural induction; a rather long Isar proof was needed. (Other complicating factors include multiset reasoning, which is not as well-developed in Isabelle/HOL as set reasoning, and the need to use ctxt_D and ctxt_D′ separately to allow Δ' to shrink while Γ' remains unchanged.)

Lemma 8.19

subjRed_op_Ref:

$[\![\, \mathsf{ctxt\_invar}\ S\ V\ T\ \Delta\ \Gamma'\ \Delta';$
$\quad h,\ \cdot,\ \Delta \Vdash \mathsf{all}\ c.\ \langle \mathsf{OpRefL}\ c\ k\ v\ (W\ c) \rangle;\ \Gamma' \vdash \langle \mathsf{Ofe}\ v\ t \rangle;\ c \notin S \,]\!] \Longrightarrow$
$\quad\quad \exists\ S_a\ V_a\ T_a.\ h,\ \cdot,\ \mathsf{CellL}\ c\ v\ \#\ \Delta \Vdash \langle \mathsf{ContL}\ k\ (\mathsf{eCell}\ c)\ (W\ c) \rangle \wedge$
$\quad\quad\quad \mathsf{ctxt\_invar}\ S_a\ V_a\ T_a\ (\mathsf{CellL}\ c\ v\ \#\ \Delta)\ (\mathsf{Ofc}\ c\ t\ \#\ \Gamma')\ (\mathsf{OfcL}\ c\ t\ \#\ \Delta')$

subjRed_op_Deref:
$[\![\, \mathsf{ctxt\_invar}\ \mathrm{S}\ \mathrm{V}\ \mathrm{T}\ \Delta\ \Gamma'\ \Delta';$
$\Gamma' \vdash \langle \mathsf{Ofc}\ \mathrm{c}\ \mathrm{t}\rangle;\ \mathsf{CellL}\ \mathrm{c}\ \mathrm{v} \in \mathsf{mset}\ \Delta\,]\!] \Longrightarrow \Gamma' \vdash \langle \mathsf{Ofe}\ \mathrm{v}\ \mathrm{t}\rangle$
subjRed_op_Assign2:
$[\![\, \mathsf{ctxt\_invar}\ \mathrm{S}\ \mathrm{V}\ \mathrm{T}\ \Delta\ \Gamma'\ \Delta';$
$\mathrm{h}, \cdot, \Delta \Vdash \mathsf{CellL}\ \mathrm{c}\ \mathrm{v} \circledcirc (\mathsf{CellL}\ \mathrm{c}\ \mathrm{v}' \multimap \mathrm{B});\ \Gamma' \vdash \langle \mathsf{Ofc}\ \mathrm{c}\ \mathrm{t}\rangle;\ \Gamma' \vdash \langle \mathsf{Ofe}\ \mathrm{v}'\ \mathrm{t}\rangle\,]\!] \Longrightarrow$
$\exists\ \mathrm{V_a}\ \Delta_1.\ \mathsf{ctxt\_invar}\ \mathrm{S}\ \mathrm{V_a}\ \mathrm{T}\ \Delta_1\ \Gamma'\ \Delta' \wedge \mathrm{h}, \cdot, \Delta_1 \Vdash \mathrm{B}$
subjRed_op_Done:
$[\![\, \mathsf{ctxt\_invar}\ \mathrm{S}\ \mathrm{V}\ \mathrm{T}\ \Delta\ \Gamma'\ \Delta';\ \cdot, \Delta \vdash \langle \mathsf{PackStateL}\ \mathrm{s}\rangle;\ \Gamma' \vdash \langle \mathsf{Ofe}\ \mathrm{v}\ \mathrm{t}\rangle\,]\!] \Longrightarrow$
$\Gamma', \Delta' \vdash \langle \mathsf{OffL}\ (\mathsf{fVal}\ \mathrm{s}\ \mathrm{v})\ \mathrm{t}\rangle$

These four lemmas provide key steps in the corresponding cases of the proof of subject reduction. None of them require induction, though two of them use SL statements with ordinal arguments for the benefit of the transfinite-induction proof of subject reduction.

The lemma subjRed_op_Deref corresponds to ofsL_s_lookup from Lemma 7.31, and follows easily from the context invariant (Definition 8.13), specifically from type_correct. The lemma subjRed_op_Assign2 corresponds to ofsL_s_assign, although its conclusion is stated with existentially quantified variables where the lemma ofsL_s_assign was more specific. It was proved with the help of Lemma 8.17 (ctxt_value_subst).

The lemma subjRed_op_Done is an easy corollary of PackStateL_OfsL that covers most of the corresponding case of subject reduction. The lemma subjRed_op_Ref was proved with the help of ctxt_invar_Cons_I, and handles the actual insertion of a new reference cell into the context, but not the HOAS aspects of the corresponding case. Neither of these lemmas corresponds directly to a lemma in Chapter 7.

Lemma 8.20 (Subject reduction with context invariant)
$[\![\, \mathsf{ctxt\_invar}\ \mathrm{S}\ \mathrm{V}\ \mathrm{T}\ \Delta\ \Gamma'\ \Delta';$
$\cdot, \Delta \vdash \langle \mathsf{EvalL}\ \mathrm{k}\ \mathrm{e}\ \mathrm{w}\rangle;\ \Gamma' \vdash \langle \mathsf{Ofe}\ \mathrm{e}\ \mathrm{t}\rangle;\ \Gamma' \vdash \langle \mathsf{Ofk}\ \mathrm{k}\ \mathrm{t}\ \mathrm{r}\rangle\,]\!] \Longrightarrow \Gamma', \Delta' \vdash \langle \mathsf{OffL}\ \mathrm{w}\ \mathrm{r}\rangle$

Proof. Unfolding Definition 8.5 in the EvalL premise, and adding a ContL case to strengthen the induction hypothesis, we obtain h :: *ordinal* such that

$$\mathrm{h}, \cdot, \Delta \Vdash \langle \mathsf{EvalL}\ \mathrm{k}\ \mathrm{e}\ \mathrm{w} \rangle \vee \mathrm{h}, \cdot, \Delta \Vdash \langle \mathsf{ContL}\ \mathrm{k}\ \mathrm{e}\ \mathrm{w} \rangle.$$

The proof then proceeds by complete induction on h.

The EvalL or ContL statement must have been derived by bc′, since the context invariant excludes these atoms from Δ; letting h′ be the derivation height of the premise, we have $\mathrm{h}' < \mathrm{h}$. Inverting Definition 8.8, there are 37 cases, one for each evaluation rule. Most cases are straightforward, with the nontrivial parts of the reasoning similar to previous formalizations; yet few were proved automatically, unlike Theorem 7.32, at least in part because of the change from structural induction to height induction. The details will not be given here. □

Lemma 8.20, with its context-invariant premise, is not a satisfactory subject-reduction theorem. Not only is the predicate ctxt_invar needlessly complicated for stating such a theorem, but it also amounts to exposing an "implementation detail". It should be possible to *state* subject reduction immediately after Section 8.2, with only simple auxiliary definitions.

Thus, the subject-reduction theorem will replace the ctxt_invar premise of this lemma with other premises. The typing information contained in the locale axiom type_correct will be derived from the SL's typing judgment for states, OfsL (Definition 7.12). This will require the use of PackStateL (Definition 8.9) to translate the SL context into a term of type *state*.

It will not be possible to entirely eliminate the use of meta-level predicates on SL contexts, because arbitrary SL contexts could contain defined SL atoms and in effect claim them without proof, making them meaningless as premises.

Definition 8.21 (Context validity predicates)

```
fun valid_L_elt :: atm ⇒ bool
  where valid_L_elt (CellL c v) = True
  | valid_L_elt (OfcL c t) = True
  | valid_L_elt _ = False
fun valid_L_ctxt :: atm list ⇒ bool
  where valid_L_ctxt · = True
  | valid_L_ctxt (h # t) = valid_L_elt h ∧ valid_L_ctxt t
definition valid_J_typing_ctxt Γ′ Δ′ =
  (∀ a. count (mset Γ′) a =
    (case a of Ofc c t ⇒ count (mset Δ′) (OfcL c t) | _ ⇒ 0))
```

The predicate valid_L_elt specifies what atoms may occur in SL linear contexts: CellL atoms and OfcL atoms. It is left to SL predicates (e.g., OfsL and PackStateL) to specify which atoms may occur in the evaluation context vs. the typing context. The predicate valid_L_ctxt simply maps valid_L_elt over lists.

The predicate valid_J_typing_ctxt is more complicated, because SL predicates cannot check non-monotonic properties of the intuitionistic context. For this reason, valid_J_typing_ctxt must check not only that the intuitionistic typing context consists of Ofc atoms, but also that they correspond one-to-one with OfcL atoms in the linear typing context. It is defined using mset and a case construct, much like ctxt_invar.

Lemma 8.22

PackStateL_OfsL_ctxt:

```
⟦ ·, Δ ⊢ ⟨PackStateL s⟩; Γ′, Δ′ ⊢ ⟨OfsL s⟩;
  valid_L_ctxt Δ; valid_L_ctxt Δ′; valid_J_typing_ctxt Γ′ Δ′ ⟧ ⟹
    ∃ S V T. ctxt_invar S V T Δ Γ′ Δ′
```

The lemma PackStateL_OfsL_ctxt "bootstraps" the context invariant from OfsL and PackStateL premises together with the context validity predicates of Definition 8.21. It is the only lemma in this formalization that has a context invariant as its conclusion but none among its premises.

It was proved by transfinite induction on the PackStateL premise. There were far fewer cases and auxiliary lemmas than the proof of subject reduction, but the Isar proof text was nonetheless long compared with the other auxiliary lemmas; it was more technical and tedious rather than nontrivial, with multiset reasoning, a complicated induction hypothesis, etc. being responsible for much of its size.

Theorem 8.23 (Subject reduction)

$[\![\cdot, \Delta \vdash \langle \mathsf{EvalL}\ \mathrm{k}\ \mathrm{e}\ \mathrm{w}\rangle; \Gamma' \vdash \langle \mathsf{Ofe}\ \mathrm{e}\ \mathrm{t}\rangle; \Gamma' \vdash \langle \mathsf{Ofk}\ \mathrm{k}\ \mathrm{t}\ \mathrm{r}\rangle;$
$\cdot, \Delta \vdash \langle \mathsf{PackStateL}\ \mathrm{s}\rangle; \Gamma', \Delta' \vdash \langle \mathsf{OfsL}\ \mathrm{s}\rangle;$
$\mathsf{valid\_L\_ctxt}\ \Delta; \mathsf{valid\_L\_ctxt}\ \Delta'; \mathsf{valid\_J\_typing\_ctxt}\ \Gamma'\ \Delta']\!] \Longrightarrow \Gamma', \Delta' \vdash \langle \mathsf{OffL}\ \mathrm{w}\ \mathrm{r}\rangle$

Proof. From the PackStateL and OfsL premises together with the context validity premises, we obtain ($\mathsf{ctxt\_invar}$ S V T Δ Γ' Δ'), for some S, V, and T, by Lemma 8.22. Together with the EvalL, Ofe, and Ofk premises, we may then apply Lemma 8.20 to deduce Γ', $\Delta' \vdash \langle \mathsf{OffL}\ \mathrm{w}\ \mathrm{r}\rangle$, as was to be proven. □

Chapter 9

Case Study:
Ordered Specification Logic

The final formalization adds ordered-logic features to the specification logic. The main objective of this change is to represent continuations in the SL context, treating their stack structure as another *logical* feature of the OL to be abstracted by the use of an SL.

This formalization was constructed by modifying the previous two-level linear formalization from Chapter 8. Most of the syntax and typing judgments are reused without modification, except for the parts relating to continuations (*cont* and Ofk).

9.1 Syntax

To represent continuations in the SL context, it is necessary to define syntax for individual elements of a continuation, which will be called *instructions*. They will be represented as Hybrid terms as in Section 5.1.

Definition 9.1 (Instructions)

types

$insn = con\ expr$

constdefs

iCons, iApp :: $(exp \Rightarrow exp) \Longrightarrow insn$

iCons E ≡ CON c_iCons $$ LAM E (iApp *similar*)

iArg, iPair1, iPair, iAssign1 :: $exp \Longrightarrow insn$

iArg e ≡ CON c_iArg $$ e (*others similar*)

iPred, iIsZero, iFst, iSnd, iRef, iDeref :: $insn$

iPred ≡ CON c_iPred (*others similar*)

iIfThen :: $[\, exp,\ exp \,] \Longrightarrow insn$

iIfThen tt ff ≡ CON c_iIfThen $$ tt $$ ff

iAssign :: $cell \Longrightarrow insn$

iAssign c ≡ CON c_iAssign $$ c

The constructors for *insn* correspond directly to those for *cont* in Definition 5.5, without the recursive argument k, except that there is no constructor corresponding to the base case kDone, which will be represented as an empty list of instructions.

9.2 Specification logic

The specification logic presented here is from [21, § 5.1], with some exceptions as in the previous formalizations: we allow an arbitrary Isabelle/HOL definition for prog in place of a specific syntax of clauses, we replace the natural-number argument used as an induction measure with an ordinal argument, and we include multiplicative connectives that are foreign to the usual logic-programming style of SLs. We also include the linear context from Sections 7.1 and 8.1, and include both left and right ordered implications, though we only use left ordered implication in formalizing the object logic.

The type of logical formulas, *prp*, extends Definition 7.1 with new connectives:

Definition 9.2

datatype $a\ prp =$

at a	$(notation\ \langle \mathrm{A} \rangle)$
ttA	$(notation\ \top)$
ttM	$(notation\ \mathbf{1})$
conjA $(a\ prp)\ (a\ prp)$	$(notation\ \mathrm{B}\ \&\ \mathrm{C})$
conjM $(a\ prp)\ (a\ prp)$	$(notation\ \mathrm{B} \otimes \mathrm{C})$
impJ $a\ (a\ prp)$	$(notation\ \mathrm{A} \supset \mathrm{B})$
impL $a\ (a\ prp)$	$(notation\ \mathrm{A} \multimap \mathrm{B})$
impO_l $a\ (a\ prp)$	$(notation\ \mathrm{A} \rightarrowtail \mathrm{B})$
impO_r $a\ (a\ prp)$	$(notation\ \mathrm{A} \twoheadrightarrow \mathrm{B})$
forall $(exp \Rightarrow a\ prp)$	$(notation$ all x. $\mathrm{B})$

The ordered SL has all the connectives of the linear SL, plus two new implications ($\rightarrowtail$ and $\twoheadrightarrow$) that will discharge assumptions from opposite ends of the ordered context.

SL statements take the form of a predicate hSL_entails with an ordinal argument, as in Definition 8.1. For this formalization there are three contexts: the intuitionistic and linear contexts seen previously, and an *ordered context* Ω. The ordered context will admit no structural rules at all.

Definition 9.3 (SL sequent rules)

class ATM = TYPE + **fixes** prog :: $[\, a,\ a\ prp \,] \Rightarrow bool$ $\quad (notation\ \mathrm{A} \leftarrow \mathrm{B})$

inductive hSL_entails :: $[\, ordinal,\ (a :: \mathrm{ATM})\ list,\ a\ list,\ alist,\ a\ prp \,] \Rightarrow bool$
$\quad (notation\ \mathrm{h},\ \Gamma,\ \Delta,\ \Omega \Vdash \mathrm{B})$

where $\mathrm{bc}' \colon \mathrm{h},\ \Gamma,\ \Delta,\ \Omega \Vdash \langle \mathrm{A} \rangle \Longleftarrow [\![\, \mathrm{A} \leftarrow \mathrm{B};\ \mathrm{h}' < \mathrm{h};\ \mathrm{h}',\ \Gamma,\ \Delta,\ \Omega \Vdash \mathrm{B} \,]\!]$

	$\mathrm{axJ}' \colon \mathrm{h},\ \Gamma,\ \cdot,\ \cdot \Vdash \langle \mathrm{A} \rangle \Longleftarrow \mathrm{A} \in \mathsf{set}\ \Gamma$
	$\mathrm{axL}' \colon \mathrm{h},\ \Gamma,\ [\,\mathrm{A}\,],\ \cdot \Vdash \langle \mathrm{A} \rangle$
	$\mathrm{axO}' \colon \mathrm{h},\ \Gamma,\ \cdot,\ [\,\mathrm{A}\,] \Vdash \langle \mathrm{A} \rangle$
	$\mathrm{ttA\_i}' \colon \mathrm{h},\ \Gamma,\ \Delta,\ \Omega \Vdash \top$
	$\mathrm{ttM\_i}' \colon \mathrm{h},\ \Gamma,\ \cdot,\ \cdot \Vdash \mathbf{1}$
	$\mathrm{conjA\_i}' \colon \mathrm{h},\ \Gamma,\ \Delta,\ \Omega \Vdash \mathrm{B}\ \&\ \mathrm{C} \Longleftarrow [\![\, \mathrm{h},\ \Gamma,\ \Delta,\ \Omega \Vdash \mathrm{B};\ \mathrm{h},\ \Gamma,\ \Delta,\ \Omega \Vdash \mathrm{C} \,]\!]$
	$\mathrm{conjM\_i}' \colon \mathrm{h},\ \Gamma,\ \Delta,\ \Omega \Vdash \mathrm{B} \otimes \mathrm{C} \Longleftarrow [\![\, \mathrm{h},\ \Gamma,\ \Delta_{\mathrm{L}},\ \Omega_{\mathrm{L}} \Vdash \mathrm{B};\ \mathrm{h},\ \Gamma,\ \Delta_{\mathrm{R}},\ \Omega_{\mathrm{R}} \Vdash \mathrm{C};$ $\mathsf{mset}\ \Delta = \mathsf{mset}\ \Delta_{\mathrm{L}} + \mathsf{mset}\ \Delta_{\mathrm{R}};$ $\Omega = \Omega_{\mathrm{L}}\ @\ \Omega_{\mathrm{R}}\ ]\!]$

| impJ_i′: h, Γ, Δ, Ω ⊩ A ⊃ B ⟸ h, A # Γ, Δ, Ω ⊩ B
| impL_i′: h, Γ, Δ, Ω ⊩ A ⊸ B ⟸ h, Γ, A # Δ, Ω ⊩ B
| impO_l_i′: h, Γ, Δ, Ω ⊩ A ⤖ B ⟸ h, Γ, Δ, A # Ω ⊩ B
| impO_r_i′: h, Γ, Δ, Ω ⊩ A ↠ B ⟸ h, Γ, Δ, Ω @ [A] ⊩ B
| all_i′: h, Γ, Δ, Ω ⊩ **all** x. B x ⟸ (⋀ x. h, Γ, Δ, Ω ⊩ B x)

The additive/multiplicative distinction applies to the ordered context as well as the linear context, but in the multiplicative case, the ordered contexts of the premises are concatenated as lists, without allowing rearrangement as is done for the linear context.

The rules directly involving contexts now have a third possibility, the ordered context. There are thus three axiom rules, one for each context. However, there are *four* implication rules, because the ordered context brings yet another distinction: where in the ordered context the discharged assumption is located. There are two reasonable possibilities, the left end or the right end; thus we have two rules impO_l_i′ and impO_r_i′, with their respective implication connectives ⤖ and ↠.

Lemma 9.4

hSL_structural:
⟦ h, Γ, Δ, Ω ⊩ B;
h ⩽ h′; **set** Γ ⊆ **set** Γ′; **mset** Δ = **mset** Δ′ ⟧ ⟹ h′, Γ′, Δ′, Ω ⊩ B

With no structural rules for the ordered context, the lemma hSL_structural has the same ordered context Ω in both premise and conclusion. For the other contexts it is the same as Lemma 8.2.

Lemma 9.5

hSL_cutJ:
⟦ h_B, A # Γ, Δ, Ω ⊩ B; h_A, Γ ⊩ ⟨A⟩ ⟧ ⟹ h_A + h_B, Γ, Δ, Ω ⊩ B
hSL_cutL:
⟦ h_B, Γ, A # $Δ_1$, Ω ⊩ B;
h_A, Γ, $Δ_2$ ⊩ ⟨A⟩;
mset Δ = **mset** $Δ_1$ + **mset** $Δ_2$ ⟧ ⟹ h_A + h_B, Γ, Δ, Ω ⊩ B

hSL_cutO:
$[\![\mathrm{h_B}, \Gamma, \Delta_1, \Omega_1 \mathbin{@} \mathrm{A} \mathbin{\#} \Omega_3 \Vdash \mathrm{B};$
$\mathrm{h_A}, \Gamma, \Delta_2, \Omega_2 \Vdash \langle \mathrm{A} \rangle;$
$\mathsf{mset}\ \Delta = \mathsf{mset}\ \Delta_1 + \mathsf{mset}\ \Delta_2]\!] \Longrightarrow \mathrm{h_A} + \mathrm{h_B}, \Gamma, \Delta, \Omega_1 \mathbin{@} \Omega_2 \mathbin{@} \Omega_3 \Vdash \mathrm{B}$

There are now three derived cut rules, one for each context. For a cut on a formula in the intuitionistic context, the cut formula must be derived with empty linear and ordered contexts; for a cut on a formula in the linear context, it must be derived with an empty ordered context; and for a cut on a formula in the ordered context, all three contexts may be nonempty.

Definition 7.6 (J) is again reused without modification; operationally, it serves to empty out the ordered context as well as the linear context. Definition 8.4 (LCONS, $\circledcirc$) is also reused without modification; operationally, its intro and elim rules are still set up to remove an atom from the linear context (never the ordered context).

Definition 9.6

$\mathrm{A} \circledcirc \mathrm{B} \equiv \langle \mathrm{A} \rangle \otimes \mathrm{B}$

The derived connective $\circledcirc$ (called OCONS in the formal theory), though definitionally identical to $\circledcirc$, is used operationally as the reverse of $\rightarrowtail$: that is, to remove the atom A from the left end of the ordered context. Intro and elim rules are provided as for $\circledcirc$, so that Isabelle's automatic proof methods will assume that $\circledcirc$ is used as stated here (and as a consequence, will fail if it is used otherwise).

Definition 9.7 (SL statements without h**)**

SL_entails :: $[(a :: \textsc{atm})\ list, a\ list, a\ list, a\ prp] \Rightarrow bool$
$(notation\ \Gamma, \Delta, \Omega \vdash \mathrm{B})$
where $(\Gamma, \Delta, \Omega \vdash \mathrm{B}) = (\exists\, \mathrm{h}.\ (\mathrm{h}, \Gamma, \Delta, \Omega \Vdash \mathrm{B}))$

As in Section 8.1, the rules and lemmas of the SL, as well as a case-analysis lemma, are all derived in this form without an ordinal argument. The rule names match those of Definition 8.1 without primes, while the lemmas use an SL_ prefix in

place of the hSL_ prefix used for the versions with ordinal arguments. The process is straightforward, consisting mostly of unfolding the definition above, except for all_i whose proof is essentially the same as for Lemma 8.6; the details are omitted here.

9.3 Evaluation and typing judgments

The type of atomic formulas has 14 constructors. EvalL, ContL, OpRefL, and Ofk have lost their *cont* arguments, and will manipulate instructions in the ordered context instead; ContL was renamed StepL, and Ofk was renamed OfkL. InsnL is a new context element representing instructions. The rest of the constructors are unchanged from Definition 8.7.

(We continue to use L suffixes for atoms that use substructural features, without distinguishing between those that use the linear context and those that use the ordered context.)

Definition 9.8

datatype *atm* = EvalL *exp final* | StepL *exp final* | OpRefL *cell exp final*
| PackStateL *state* | CellL *cell exp* | InsnL *insn*
| IsTerm *exp* | Ofe *exp tp* | OfkL *tp tp* | Ofc *cell tp*
| OfcL *cell tp* | OfsL *state* | OffL *final tp* | OffNewL *cell final tp*

In addition to using the intuitionistic and linear contexts as in Section 8.2, some of the evaluation judgments (EvalL, StepL, and OpRefL but not PackStateL) use the ordered context to represent a stack of instructions (in the form of InsnL atoms), i.e., a continuation. The typing judgment for continuations, OfkL, also uses the ordered context for this purpose. (It functions as a fairly direct replacement for the *cont* argument from previous formalizations.) None of the other SL predicates use the ordered context.

The specification of PackStateL (Definition 8.9) and the typing judgments other

than OfkL (Definitions 7.9, 7.10, 7.12, and 7.13) are unchanged from previous formalizations.

Definition 9.9 (Specification of SL atoms, part 1 – EvalL, StepL, OpRefL)

inductive prog_atm :: $[\, atm,\ atm\ prp \,] \Rightarrow bool$ $\quad$ ($notation$ A ← B)

where ev_Zero_I: EvalL eZero w ← ⟨StepL eZero w⟩

| ev_Suc_I: EvalL (eSuc e) w ← InsnL (iCons (λ v. eSuc v)) ⤖ ⟨EvalL e w⟩

| ev_Pred_I: EvalL (ePred e) w ← InsnL iPred ⤖ ⟨EvalL e w⟩

| ev_IsZero_I: EvalL (eIsZero e) w ← InsnL iIsZero ⤖ ⟨EvalL e w⟩

| ev_True_I: EvalL eTrue w ← ⟨StepL eTrue w⟩

| ev_False_I: EvalL eFalse w ← ⟨StepL eFalse w⟩

| ev_IfThen_I: EvalL (eIfThen c tt ff) w ← InsnL (iIfThen tt ff) ⤖ ⟨EvalL c w⟩

| ev_Unit_I: EvalL eUnit w ← ⟨StepL eUnit w⟩

| ev_Pair_I: EvalL (ePair x y) w ← InsnL (iPair1 y) ⤖ ⟨EvalL x w⟩

| ev_Fst_I: EvalL (eFst p) w ← InsnL iFst ⤖ ⟨EvalL p w⟩

| ev_Snd_I: EvalL (eSnd p) w ← InsnL iSnd ⤖ ⟨EvalL p w⟩

| ev_App_I: EvalL (f $ x) w ← InsnL (iArg x) ⤖ ⟨EvalL f w⟩

| ev_Fn_I: EvalL (eFn F) w ← ⟨StepL (eFn F) w⟩ ⟸ abstr F

| ev_Fix_I: EvalL (eFix F) w ← ⟨EvalL (F (eFix F)) w⟩ ⟸ abstr F

| ev_Letv_I: EvalL (eLetv v F) w ← ⟨EvalL (F v) w⟩ ⟸ abstr F

| ev_Ref_I: EvalL (eRef e) w ← InsnL iRef ⤖ ⟨EvalL e w⟩

| ev_Deref_I: EvalL (eDeref e) w ← InsnL iDeref ⤖ ⟨EvalL e w⟩

| ev_Assign_I: EvalL (eAssign m e) w ← InsnL (iAssign1 e) ⤖ ⟨EvalL m w⟩

| ev_Cell_I: EvalL (eCell c) w ← ⟨StepL (eCell c) w⟩

| op_Cons_I: StepL v w ← InsnL (iCons E) ⊛ ⟨StepL (E v) w⟩ ⟸ abstr E

| op_Arg_I: StepL (eFn F) w
← InsnL (iArg x) ⊛ InsnL (iApp F) ⤖ ⟨EvalL x w⟩ ⟸ abstr F

| op_App_I: StepL v w ← InsnL (iApp E) ⊛ ⟨EvalL (E v) w⟩ ⟸ abstr E

| op_Pred_I1: StepL eZero w ← InsnL iPred ⊛ ⟨StepL eZero w⟩

| op_Pred_I2: StepL (eSuc v) w ← InsnL iPred ⊛ ⟨StepL v w⟩

| op_IsZero_I1: StepL eZero w ← InsnL iIsZero ⊛ ⟨StepL eTrue w⟩

| op_IsZero_I2: StepL (eSuc v) w ← InsnL iIsZero ⊛ ⟨StepL eFalse w⟩

| op_IfThen_I1: StepL eTrue w ← InsnL (iIfThen tt ff) ⊛ ⟨EvalL tt w⟩

| op_IfThen_I2: StepL eFalse w ← InsnL (iIfThen tt ff) ⊛ ⟨EvalL ff w⟩

$$
\begin{array}{l}
\mid \text{op\_Pair1\_I}\colon \mathsf{StepL}\ \mathrm{v}\ \mathrm{w} \leftarrow \mathsf{InsnL}\ (\mathsf{iPair1}\ \mathrm{y}) \otimes \mathsf{InsnL}\ (\mathsf{iPair}\ \mathrm{v}) \rightarrowtail \langle \mathsf{EvalL}\ \mathrm{y}\ \mathrm{w}\rangle \\
\mid \text{op\_Pair\_I}\colon \mathsf{StepL}\ \mathrm{v'}\ \mathrm{w} \leftarrow \mathsf{InsnL}\ (\mathsf{iPair}\ \mathrm{v}) \otimes \langle \mathsf{StepL}\ (\mathsf{ePair}\ \mathrm{v}\ \mathrm{v'})\ \mathrm{w}\rangle \\
\mid \text{op\_Fst\_I}\colon \mathsf{StepL}\ (\mathsf{ePair}\ \mathrm{v}\ \mathrm{v'})\ \mathrm{w} \leftarrow \mathsf{InsnL}\ \mathsf{iFst} \otimes \langle \mathsf{StepL}\ \mathrm{v}\ \mathrm{w}\rangle \\
\mid \text{op\_Snd\_I}\colon \mathsf{StepL}\ (\mathsf{ePair}\ \mathrm{v}\ \mathrm{v'})\ \mathrm{w} \leftarrow \mathsf{InsnL}\ \mathsf{iSnd} \otimes \langle \mathsf{StepL}\ \mathrm{v'}\ \mathrm{w}\rangle \\
\mid \text{op\_Ref\_I}\colon \mathsf{StepL}\ \mathrm{v}\ (\mathsf{fNew}\ \mathrm{W}) \\
\qquad\qquad \leftarrow \mathsf{InsnL}\ \mathsf{iRef} \otimes (\mathsf{all}\ \mathrm{c}.\ \langle \mathsf{OpRefL}\ \mathrm{c}\ \mathrm{v}\ (\mathrm{W}\ \mathrm{c})\rangle) \Longleftarrow \mathsf{abstr}\ \mathrm{W} \\
\mid \text{op\_Deref\_I}\colon \mathsf{StepL}\ (\mathsf{eCell}\ \mathrm{c})\ \mathrm{w} \\
\qquad\qquad\quad \leftarrow \mathsf{InsnL}\ \mathsf{iDeref} \otimes (\mathsf{CellL}\ \mathrm{c}\ \mathrm{v} \otimes \top)\ \&\ \langle \mathsf{StepL}\ \mathrm{v}\ \mathrm{w}\rangle \\
\mid \text{op\_Assign1\_I}\colon \mathsf{StepL}\ (\mathsf{eCell}\ \mathrm{c})\ \mathrm{w} \\
\qquad\qquad\quad\ \leftarrow \mathsf{InsnL}\ (\mathsf{iAssign1}\ \mathrm{e}) \otimes \mathsf{InsnL}\ (\mathsf{iAssign}\ \mathrm{c}) \rightarrowtail \langle \mathsf{EvalL}\ \mathrm{e}\ \mathrm{w}\rangle \\
\mid \text{op\_Assign\_I}\colon \mathsf{StepL}\ \mathrm{v}\ \mathrm{w} \\
\qquad\qquad\quad \leftarrow \mathsf{InsnL}\ (\mathsf{iAssign}\ \mathrm{c}) \otimes \mathsf{CellL}\ \mathrm{c}\ \mathrm{v'} \otimes (\mathsf{CellL}\ \mathrm{c}\ \mathrm{v} \multimap \langle \mathsf{StepL}\ \mathrm{v}\ \mathrm{w}\rangle) \\
\mid \text{op\_Done\_I}\colon \mathsf{StepL}\ \mathrm{v}\ (\mathsf{fVal}\ \mathrm{s}\ \mathrm{v}) \leftarrow \langle \mathsf{PackStateL}\ \mathrm{s}\rangle \\
\mid \text{opRefL\_I1}\colon \mathsf{OpRefL}\ \mathrm{c}\ \mathrm{v}\ \mathrm{w} \leftarrow \mathsf{CellL}\ \mathrm{c}\ \mathrm{v'} \otimes \top \\
\mid \text{opRefL\_I}\colon \mathsf{OpRefL}\ \mathrm{c}\ \mathrm{v}\ \mathrm{w} \leftarrow \mathsf{CellL}\ \mathrm{c}\ \mathrm{v} \multimap \langle \mathsf{StepL}\ (\mathsf{eCell}\ \mathrm{c})\ \mathrm{w}\rangle
\end{array}
$$

All of the evaluation rules correspond straightforwardly to those of Definition 8.8, by replacing construction and matching of continuations with insertion and matching/removal of instructions in the ordered context using the connectives $\rightarrowtail$ and $\otimes$ respectively.

Definition 9.10 (Specification of SL atoms, part 2 – OfkL)

inductive prog_atm :: $\left[\, atm,\ atm\ prp \,\right] \Rightarrow bool$ $\quad$ ($notation$ A $\leftarrow$ B)

where ofkL_Cons_I:

$$
\begin{array}{l}
\quad \mathsf{OfkL}\ \mathrm{t}\ \mathrm{r} \leftarrow \mathsf{InsnL}\ (\mathsf{iCons}\ \mathrm{E}) \otimes \mathsf{J}\ \langle \mathsf{Ofe}\ (\mathsf{eFn}\ \mathrm{E})\ (\mathsf{tFun}\ \mathrm{t}\ \mathrm{s})\rangle\ \&\ \langle \mathsf{OfkL}\ \mathrm{s}\ \mathrm{r}\rangle \\
\qquad\qquad \Longleftarrow \mathsf{abstr}\ \mathrm{E} \\
\mid \text{ofkL\_Arg\_I}\colon \mathsf{OfkL}\ (\mathsf{tFun}\ \mathrm{t}\ \mathrm{s})\ \mathrm{r} \leftarrow \mathsf{InsnL}\ (\mathsf{iArg}\ \mathrm{x}) \otimes \mathsf{J}\ \langle \mathsf{Ofe}\ \mathrm{x}\ \mathrm{t}\rangle\ \&\ \langle \mathsf{OfkL}\ \mathrm{s}\ \mathrm{r}\rangle \\
\mid \text{ofkL\_App\_I}\colon \\
\quad \mathsf{OfkL}\ \mathrm{t}\ \mathrm{r} \leftarrow \mathsf{InsnL}\ (\mathsf{iApp}\ \mathrm{E}) \otimes \mathsf{J}\ \langle \mathsf{Ofe}\ (\mathsf{eFn}\ \mathrm{E})\ (\mathsf{tFun}\ \mathrm{t}\ \mathrm{s})\rangle\ \&\ \langle \mathsf{OfkL}\ \mathrm{s}\ \mathrm{r}\rangle \\
\qquad\qquad \Longleftarrow \mathsf{abstr}\ \mathrm{E} \\
\mid \text{ofkL\_Pred\_I}\colon \mathsf{OfkL}\ \mathsf{tNat}\ \mathrm{r} \leftarrow \mathsf{InsnL}\ \mathsf{iPred} \otimes \langle \mathsf{OfkL}\ \mathsf{tNat}\ \mathrm{r}\rangle \\
\mid \text{ofkL\_IsZero\_I}\colon \mathsf{OfkL}\ \mathsf{tNat}\ \mathrm{r} \leftarrow \mathsf{InsnL}\ \mathsf{iIsZero} \otimes \langle \mathsf{OfkL}\ \mathsf{tBool}\ \mathrm{r}\rangle \\
\mid \text{ofkL\_IfThen\_I}\colon \\
\quad \mathsf{OfkL}\ \mathsf{tBool}\ \mathrm{r} \leftarrow \mathsf{InsnL}\ (\mathsf{iIfThen}\ \mathrm{tt}\ \mathrm{ff}) \otimes \mathsf{J}\ \langle \mathsf{Ofe}\ \mathrm{tt}\ \mathrm{t}\rangle\ \&\ \mathsf{J}\ \langle \mathsf{Ofe}\ \mathrm{ff}\ \mathrm{t}\rangle\ \&\ \langle \mathsf{OfkL}\ \mathrm{t}\ \mathrm{r}\rangle
\end{array}
$$

| ofkL_Pair1_I: OfkL t r ← InsnL (iPair1 y) ⊛ J ⟨Ofe y s⟩ & ⟨OfkL (tPair t s) r⟩
| ofkL_Pair_I: OfkL s r ← InsnL (iPair v) ⊛ J ⟨Ofe v t⟩ & ⟨OfkL (tPair t s) r⟩
| ofkL_Fst_I: OfkL (tPair t s) r ← InsnL iFst ⊛ ⟨OfkL t r⟩
| ofkL_Snd_I: OfkL (tPair t s) r ← InsnL iSnd ⊛ ⟨OfkL s r⟩
| ofkL_Ref_I: OfkL t r ← InsnL iRef ⊛ ⟨OfkL (tRef t) r⟩
| ofkL_Deref_I: OfkL (tRef t) r ← InsnL iDeref ⊛ ⟨OfkL t r⟩
| ofkL_Assign1_I: OfkL (tRef t) r ← InsnL (iAssign1 e) ⊛ J ⟨Ofe e t⟩ & ⟨OfkL t r⟩
| ofkL_Assign_I: OfkL t r ← InsnL (iAssign c) ⊛ J ⟨Ofc c t⟩ & ⟨OfkL t r⟩
| ofkL_Done_I: OfkL r r ← **1**

The typing rules for continuations (as represented in the ordered SL context) also correspond straightforwardly to those of Definition 7.11 (for continuations represented as Hybrid terms of type *cont*), in the same way as for the evaluation rules, except that instructions are only ever matched and removed, never inserted, in the ordered context.

9.4 Context invariants

The partial context invariants ctxt_abstr, ctxt_D, ctxt_G′, and ctxt_D′ from Definitions 8.10 to 8.12 were reused without modification.

Definition 9.11

locale ctxt_Z =
fixes Ω :: *atm list*
assumes Z_mem: a mem Ω ⟹ (case a of InsnL i ⇒ True | _ ⇒ False)

The locale ctxt_Z merely requires that the ordered context Ω consist of InsnL atoms. This condition is sufficient to apply the typing judgment OfkL, which imposes more specific conditions on the ordered context.

There is no dependency on ctxt_abstr, indeed the predicate ctxt_Z is simple enough that it would be an acceptable premise for subject reduction.

Definition 9.12

locale ctxt_invar = ctxt_D + ctxt_G′ + ctxt_D′ + ctxt_Z +
assumes type_correct: c ∈ S ⟹ Γ′ ⊢ ⟨Ofe (V c) (T c)⟩

The locale ctxt_invar adds ctxt_Z, with its ordered context Ω, to Definition 8.13. This results in a predicate (ctxt_invar S V T Δ Γ′ Δ′ Ω) with seven arguments. There is no interaction between the ordered context and the other locale parameters.

Lemmas 8.14 to 8.17 (properties of the context invariant) were reused with only one trivial change, the addition of an argument Ω to each instance of the predicate ctxt_invar. Additional inversion lemmas were proved for SL statements involving the ordered context Ω, as well as lemmas allowing the addition or removal of instructions in Ω while preserving the context invariant. These were all straightforward and their statements are omitted here.

9.5 Semantic lemmas and subject reduction

Lemmas 8.18 (PackStateL_OfsL) and 8.19 (subjRed_op_Ref, subjRed_op_Deref, subjRed_op_Assign2, and subjRed_op_Done) were reused with only trivial changes: the addition of an Ω argument to each instance of ctxt_invar, and the addition of Ω as ordered context in each SL statement involving the evaluation context Δ, except for the PackStateL premise of PackStateL_OfsL.

Lemma 9.13

PackStateL_Z_empty:
⟦ ctxt_Z Ω; h, ·, Δ, Ω ⊩ ⟨PackStateL s⟩ ⟧ ⟹ Ω = ·

This lemma was proved by an easy transfinite induction on h; it shows that a PackStateL statement implies that its ordered context is empty. This ensures that a StepL statement can be derived starting with op_Done_I only when there

are no instructions remaining to be executed. (It is also the reason why the lemma PackStateL_OfsL could be reused without adding an Ω argument.)

Unusually, this lemma requires no validity condition on Δ, since the axL′ rule (Definition 9.3) cannot apply unless Ω is empty as required.

Lemma 9.14 (Subject reduction with context invariant)

$[\![\, \mathsf{ctxt\_invar}\ \mathrm{S}\ \mathrm{V}\ \mathrm{T}\ \Delta\ \Gamma'\ \Delta'\ \Omega;$
$\cdot, \Delta, \Omega \vdash \langle \mathsf{EvalL}\ \mathrm{e}\ \mathrm{w}\rangle;$
$\Gamma' \vdash \langle \mathsf{Ofe}\ \mathrm{e}\ \mathrm{t}\rangle; \Gamma', \cdot, \Omega \vdash \langle \mathsf{OfkL}\ \mathrm{t}\ \mathrm{r}\rangle\,]\!] \Longrightarrow \Gamma', \Delta' \vdash \langle \mathsf{OffL}\ \mathrm{w}\ \mathrm{r}\rangle$

The proof of this subject reduction lemma was surprisingly similar to that of the corresponding lemma from the previous formalization (Lemma 8.20). The only significant differences were the replacement of matching and construction of Hybrid terms of type *cont* with manipulation of the ordered context using the connectives $\circledcirc$ and $\rightarrowtail$, requiring different patterns of inversion and application of rules, and the need to use the lemma PackStateL_Z_empty in the case op_Done_I in place of matching the base case kDone.

The context validity predicates from Definition 8.21 were reused without modification; the lemma PackStateL_OfsL_ctxt (Lemma 8.22) was reused with only the addition of an *empty* ordered-context argument to ctxt_invar in the conclusion.

Theorem 9.15 (Subject reduction)

$[\![\, \cdot, \Delta \vdash \langle \mathsf{EvalL}\ \mathrm{e}\ \mathrm{w}\rangle; \Gamma' \vdash \langle \mathsf{Ofe}\ \mathrm{e}\ \mathrm{t}\rangle;$
$\cdot, \Delta \vdash \langle \mathsf{PackStateL}\ \mathrm{s}\rangle; \Gamma', \Delta' \vdash \langle \mathsf{OfsL}\ \mathrm{s}\rangle;$
$\mathsf{valid\_L\_ctxt}\ \Delta; \mathsf{valid\_L\_ctxt}\ \Delta'; \mathsf{valid\_J\_typing\_ctxt}\ \Gamma'\ \Delta'\,]\!] \Longrightarrow \Gamma', \Delta' \vdash \langle \mathsf{OffL}\ \mathrm{w}\ \mathrm{r}\rangle$

This form of subject reduction follows from Lemma 9.14 and PackStateL_OfsL_ctxt by the same reasoning used for Theorem 8.23.

The subject reduction theorem stated here is weaker than Theorem 8.23, because it does not allow a nonempty continuation. It would be straightforward to strengthen it, with the help of an additional context-validity predicate for the ordered context.

Chapter 10

Case Study: Observations and Related Work

The one-level formalization of Chapter 5 and the two-level formalization with intuitionistic SL of Chapter 6 were both reasonably concise, and successful in automating most of the cases of the subject reduction proof, except for those involving mutable references.[1]

Comparing these two formalizations, the differences start with the typing judgments, as the same representation of syntax and evaluation was used. The one-level formalization required a context argument C for typing of program variables, where the two-level formalization used hypothetical judgments, with types for program variables given by Ofe assumptions in the SL context.

The two-level intuitionistic formalization was roughly 10–20% larger that the one-level formalization (depending on how it is measured), including the definition and properties of the specification logic itself, which can be reused. Excluding the SL, the difference is much smaller, and the residual difference is largely due to the use of an awkward form of induction to prove weakening for the remaining context argument

[1]These formalizations required more detail than the corresponding *informal* treatment (Chapter 4), but that is generally true of contemporary formal proofs.

D (for location variables) in Lemma 6.13. There was some notational overhead for the use of SL statements in the two-level formalization, but it was minimal and it could be further reduced by Isabelle syntax definitions if desired. On the other hand, the one-level formalization had to prove weakening and substitution lemmas for its context argument C, where the two-level intuitionistic formalization used (reusable) properties of the SL.

The same three cases of subject reduction – dereferencing, assignment, and allocation of mutable references – required nontrivial (i.e., not fully automated) Isar proofs in both of these formalizations. (In all of the formalizations, the total number of cases contributed far less to to the complexity of the proof of subject reduction than the presence of mutable references did.)

Moving on to the formalization of Chapter 7 with its use of linear logic in the SL, we find more significant differences due to the representation of types for location variables in the SL's linear context. The typing rules that do not involve variables are now entirely free of context arguments, though typing *statements* must, of course, include an SL context. The typing rules that *do* involve variables contain SL connectives, which is a minor notational overhead in most cases, but a more significant one for the rule offL_New_I which must simulate a disjuction using an auxiliary predicate.

In the two-level intuitionistic formalization, the proof of subject reduction involved SL statements with *empty* contexts only. However, with typing of location variables in the SL context, this is no longer the case for the two-level linear formalization. This requires us to use a *context invariant*, as described in Section 7.3, to constrain the form of SL statements. The definition and properties of context invariants added significantly to the length and complexity of the formalization. The loss of a convenient form of structural induction on typing judgments (for Lemma 7.31) had a similar effect.

With a little fine-tuning (discussed in Section 10.1 below), the two-level linear

formalization achieved automatic proof of the same cases of subject reduction as the previous two formalizations, with moderately longer Isar proofs for the three cases related to mutable references.

In total, this third formalization was nearly twice the size of either of the previous formalizations, and was significantly more difficult to construct. To remedy this situation, we believe it will be necessary to provide general tools (lemmas, etc.) for working with SL contexts, as well as a form of induction on SL statements that follows the structure of the OL judgments encoded in the SL. Such improvements would change the formalization to such an extent that we cannot make an informed speculation on how well the result would compare with the formalizations of Chapters 5 and 6.

Turning now to the formalization of Chapter 8, we find the first change to the representation of evaluation, which is moved from the meta-level to the SL. This only compounded the difficulties encountered in the previous formalization, and is better viewed as an attempt to push the limits of Hybrid rather than a practical way of formalizing an object language. We had little choice but to seek a more convenient form of induction, and we found one in the use of an ordinal argument in SL statements as an induction measure in Section 8.1. This improves on the use of a natural-number argument for this purpose, by allowing even infinitely-branching SL rules to be used in a form without the induction measure. This allowed us to avoid the tedious task of tracking the induction measure for *typing* statements throughout the proof of subject reduction by induction on an *evaluation* statement, which would have been needed for the sake of a single use of all_i in the op_Ref_I case had we used a natural-number induction measure.

We also developed a modular approach to defining context invariants, based on Isabelle's **locale** mechanism, in Section 8.3. Automatic proof of the easy cases of subject reduction was lost, and the resulting Isar proof was long and tedious. The formalization was completed, but it does not represent anything close to a practical

approach; its value lies mainly in the development of useful techniques for formal proof.

Finally, we consider the formalization of Chapter 9, which added ordered logic features to the SL and used the ordered-logic context to represent continuations. Somewhat surprisingly, this had very little effect on the structure of the formalization, amounting to little more than a syntactic change as compared with the linear-SL formalization of Chapter 8.

In general, the use of Hybrid's HOAS worked well, eliminating the need for technical lemmas related to variable binding. The two-level approach was also quite usable in Chapter 6, with an intuitionistic SL and evaluation still represented at the meta-level. The formalizations based on sub-structural SLs, on the other hand, were heavily burdened with technical lemmas and proof steps related to contexts and induction; further work would be needed to determine to what extent these problems can be alleviated.

Of the improvements to Hybrid from Chapter 3, the elimination of proper and the stronger injectivity property for LAM were directly useful. The adequacy result from Section 3.4, which made use of many of the other improvements, addresses most of the adequacy issue for the case study's representations of OL syntax, although the details as well as adequacy for OL judgments remain as future work.

10.1 Proof Automation

In developing the formalizations of Mini-ML with references presented here, various techniques were used to facilitate automation of the formal proofs in Isabelle/HOL. Most of these techniques did not appear in Chapters 5 to 9, either because of the omission of proofs, or because the lemmas or definitions involved were considered too technical to present there. Some of the more notable ones are listed here:

- When making auxiliary definitions, their basic properties were stated in the form of natural-deduction style introduction and elimination rules for the use of Isabelle's classical reasoner, as well as rewrite rules for the use of Isabelle's simplifier. This is a standard technique in Isabelle/HOL, but there are some subtleties. For instance, even *trivial* elimination rules are useful, to remove meaningless premises from subgoals. Also, some care is required with simplifier rules to avoid nonterminating sequences of rewrites. These came up in surprising places, such as Definitions 6.7 and 6.7: proof of the elimination rules (not shown) involved 510 cases, all of which should have been trivial, but *one* of which required a manual proof step to prevent a simplifier loop!

- Isabelle's automatic proof methods had trouble finding even obvious instantiations, e.g., in proving existentially-quantified statements. Sometimes the use of a different proof method, such as "**best**" (named for its search strategy) rather than the more usual "**auto**" in Theorem 6.16, dealt with the problem. In other cases, such as Theorem 7.32, increasing the search depth limits was successful. However, at other times these approaches did not produce any result in a reasonable length of time. In inductive proofs, it often helped to treat the IH as an introduction rule rather than a premise.

  In the formalizations of Chapters 8 and 9, an attempt was made to solve the problem in general using a trivial predicate to suggest instantiations for the automatic proof methods: (try x) was defined equal to True but used to trigger the use of rules instantiating quantifiers with x. However, this approach met with only limited success.

  The combination of instantiation and *unsafe* intro/elim rules (for which the automatic proof methods must be prepared to backtrack), even when not in the same proof step, tended to cause particular trouble. In Chapter 7, the OL typing rules were combined with the SL's backchaining rule and declared

as safe introduction rules, and a similar combination was used to obtain safe elimination rules, which was essential for the automation of the "easy" cases of subject reduction.

- Isabelle/HOL's Multiset library was missing several basic properties that are useful in reasoning about multisets. We proved some such properties as part of the formalizations with linear and ordered SLs; even so, proofs involving multisets often caused trouble for automation. We speculate that the use of more advanced features such as *simplifier procedures* may be needed to properly support multiset reasoning in Isabelle/HOL.

10.2 Related Work

We discuss some of the related work in two main areas: formalizations of programming languages with mutable state (including functional languages with references, and imperative languages) in other systems, and formalizations of other object languages and properties in Hybrid.

Other formalizations of languages with mutable state

- An example by Simmons on the Twelf Project website [71, 72] formalizes type safety for a minimal language with mutable state. The formalization is nonetheless quite large; Felty and Momigliano [21] note that it is much larger than a similar example for (purely functional) Mini-ML.

- Cervesato and Pfenning [9] formalize Mini-ML with references in a linear logical framework. (Our object language is from that paper, as mentioned in Chapter 4.) This approach seems to be reasonably successful in avoiding the complications that arise in the Twelf example.

- Nipkow [55] formalizes several kinds of semantics for a small imperative programming language in Isabelle/HOL, proves their equivalence, and proves several other properties. State is represented using functions from locations to values. The reported sizes of the theory files are quite reasonable given the amount of metatheory covered.

It seems that the *combination* of working in a logical framework (as opposed to a proof assistant such as Isabelle/HOL) and not having linear-logic features is particularly problematic.

Other examples and case studies in Hybrid

- In their paper presenting Hybrid [2], Ambler, Crole, and Momigliano formalized two small examples: negation normal form for quantified propositional logic, and operational semantics for a lazy λ-calculus. Object-language judgments were represented directly in Isabelle/HOL (a one-level approach).

- Momigliano, Ambler, and Crole [50] prove that bisimulation for the lazy λ-calculus is a congruence. This paper also used a one-level approach, with reasoning over open terms using Hybrid's VAR.

- Momigliano and Ambler [49] prove correctness of compilation as well as subject reduction for a small fragment of Mini-ML. This paper introduces the two-level approach in Hybrid for the first time, inspired by Felty's axiomatic treatment of HOAS in Coq [19]. (We use the same specification logic in Chapter 6.)

- Felty and Momigliano [21] prove subject reduction for a fragment of Mini-ML, using the same approach that Momigliano and Ambler used [49]. They also prove subject reduction using a continuation-style operational semantics represented in an ordered SL. (We use a similar SL and similar encoding techniques in Chapter 9.)

- Felty and Momigliano [20] prove type unicity for PCF (a functional language similar to Mini-ML), using a natural-number argument as an induction measure for SL statements, and reasoning about open terms using Hybrid's VAR. PCF is a larger object language than those of previous formalizations, but it is still a purely functional language without mutable references.

These publications are listed on the Hybrid website [39], and some of them are accompanied by corresponding formal theory files.

Chapter 11

Conclusion and Future Work

11.1 Summary of Work and Results

In this thesis, we presented an improved version of Hybrid, a system for higher order abstract syntax in Isabelle/HOL. One set of improvements centered on the modification of the type of terms *expr* to exclude terms with dangling de Bruijn indices. This small change paved the way for a stronger injectivity property, which allows abstr conditions put into an inductively-defined predicate via introduction rules to be brought out again in the elimination rules, rather than required again as premises. It also facilitated the use of Isabelle's simplifier to convert between HOAS and de Bruijn indices without the use of special-purpose tactics.

We formally proved that a generalization of abstr to binary functions can be defined in terms of abstr on unary functions. A corollary of this result (abstr_LAM) enabled abstr conditions to be proved compositionally instead of by converting from HOAS to de Bruijn syntax. This corollary and another one (abstr_nchotomy) also completed Hybrid's characterization of its type *expr*, in such a way that the subsequent proof of adequacy did not need to refer to de Bruijn syntax.

Another direction of work on Hybrid concerned induction and the representation

of open terms. We generalized Hybrid's predicate abstr to n-ary functions, and proved an induction principle for such functions that maintains a HOAS representation even for open terms. This work was experimental and integration into Hybrid remains as future work.

Improving on prior work, we proved representational adequacy of Hybrid's HOAS for a λ-calculus-like fragment of Isabelle/HOL syntax. This result was stated in terms of set-theoretic semantics for higher-order logic, and thus applied directly to Hybrid as an Isabelle/HOL formal theory. It was also based solely on Hybrid's type *expr* and its properties, making the adequacy result independent of the details of Hybrid's definition of HOAS in terms of de Bruijn indices.

We also presented a case study of the formalization of a small programming language, Mini-ML with references, in Hybrid. We developed five variant formalizations of the operational semantics, type system, and subject reduction theorem for this language, using the two-level approach with various specification logics as well as a one-level formalization for comparison purposes. Both the one-level approach and the two-level approach with intuitionistic SL were found to be reasonably successful ways of formalizing this object language, where the use of Hybrid avoided the need for technical lemmas relating to variable binding, and other overhead was minimal. Variants with sub-structural (linear and ordered) SLs were found to cause considerable difficulties with proof automation in Isabelle/HOL. While there is likely to be much room for improvement, the overhead of these approaches presently makes them unattractive for practical formalization tasks as compared with the other approaches that we tried. The process of constructing these formalizations did yield some useful insights regarding the structuring of formal proofs based on Hybrid.

11.2 Future Work

Hybrid

There are several directions in which our work on Hybrid could be continued.

One is to integrate our work on n-ary syntactic functions into Hybrid. This is a promising direction for further improvement of Hybrid, but many technical details remain to be worked out. (The use of type classes to generalize abstr to curried n-ary functions is an interesting alternative; while it does not lead to an induction principle that would be well-typed in Isabelle/HOL, it might be useful in combination with the approach of Section 3.3.)

Another is to attempt to apply some of our improvements to the versions of Hybrid for Coq. As described in [21], however, the difference between higher-order logic and dependent type theory requires some differences in approach. Some divergence between the versions of Hybrid for different proof assistants is thus likely to persist.

On the other hand, some of our improvements may actually be more effective in Coq. We could represent the hierarchy of de Bruijn levels as a *type family* indexed by natural numbers, internalizing not just the proper predicate but also the level predicate into the type system. Also, internalizing abstr as a type would be more attractive, as we could represent it using *dependent pairs* consisting of a function together with a proof that it is of the desired form.

Finally, an important objective is to extend Hybrid to a *typed* system providing an interface similar to Isabelle/HOL's **datatype** package or Urban's nominal datatype package [73]. There has been some work in this direction by Capretta and Felty [7] for Hybrid in Coq; we could attempt to adapt the ideas there to Isabelle/HOL, though again some differences in approach may be appropriate.

Case Study

While there is much room for technical improvement in the formalizations of our case study, other directions appear to be more interesting.

One such direction is to formalize in Isabelle/HOL and Hybrid properties that require the use of induction on Hybrid terms possibly containing LAM. Our work has so far avoided that form of induction, though providing it is considered an important goal of Hybrid. Proving the equivalence of HOAS representations based on Hybrid with first-order representations of the same object logics would be a good example. Some experiments in this direction inspired our work on n-ary syntactic functions in Hybrid.

However, the main direction in which we believe this work should be continued concerns adequacy. While adequacy of the representation of object-language terms should follow easily from the adequacy result that we have established for Hybrid, the representation of object-language judgments using a SL also raises questions of adequacy. A provability-preserving bijection should be established between the set of statements for each OL judgment and a set of corresponding SL statements. The latter set would be specified by a particular atom in the conclusion together with a context invariant, and it is expected that context invariants might be better explained in the context of adequacy.

Experience has shown that if such properties are not verified for a large formalization, such as those presented in this thesis, then they are not likely to be true. On several occasions, problems were found that would clearly break any form of adequacy; they were fixed, but it seems likely that similar problems may remain. However, we believe that only minor changes to the formalizations will be required for a proof of adequacy to succeed.

Another direction for future work is the development of general techniques and tools for handling contexts, perhaps along the lines of Twelf's "blocks" and "worlds".

This would further the goal of simulating Twelf's approach in the definitional setting of Isabelle/HOL, and it is expected to offer simplifications for both the formalizations and the proofs of adequacy.

Bibliography

[1] Martin Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy, *Explicit substitutions*, Journal of Functional Programming **1** (1991), no. 4, 375–416.

[2] Simon Ambler, Roy Crole, and Alberto Momigliano, *Combining higher order abstract syntax with tactical theorem proving and (co)induction*, Theorem Proving in Higher Order Logics (Victor Carreño, César Muñoz, and Sofiène Tahar, eds.), Lecture Notes in Computer Science, vol. 2410, Springer, 2002, pp. 13–30.

[3] Kenneth Appel and Wolfgang Haken, *Every planar map is four colourable*, Bulletin of the American Mathematical Society **82** (1976), 711–712.

[4] Kenneth Appel and Wolfgang Haken, *Every planar map is four colourable*, Contemporary Mathematics, vol. 98, American Mathematical Society, 1989.

[5] H. P. Barendregt, *The lambda calculus: Its syntax and semantics*, North-Holland, Amsterdam, 1984.

[6] Yves Bertot and Pierre Castéran, *Interactive theorem proving and program development. Coq'Art: The calculus of inductive constructions*, Springer, 2004.

[7] Venanzio Capretta and Amy Felty, *Higher-order abstract syntax in type theory*, Logic Colloquium 2006 (S. Barry Cooper, Herman Geuvers, Anand Pillay, and Jouko Väänänen, eds.), Lecture Notes in Logic, vol. 32, Cambridge University Press, 2009, pp. 65–90.

[8] Venanzio Capretta and Amy P. Felty, *Combining de Bruijn indices and higher-order abstract syntax in Coq*, Types for Proofs and Programs – TYPES 2006 (Thorsten Altenkirch and Conor McBride, eds.), Lecture Notes in Computer Science, vol. 4502, Springer, 2007, pp. 63–77.

[9] Iliano Cervesato and Frank Pfenning, *A linear logical framework*, Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science, 1996.

[10] Adam Chlipala, *Parametric higher-order abstract syntax for mechanized semantics*, ICFP'08: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming, ACM, September 2008.

[11] Alonzo Church, *A formulation of the simple theory of types*, Journal of Symbolic Logic **5** (1940), no. 2, 56–68.

[12] D. Clement, J. Despeyroux, T. Despeyroux, and G. Kahn, *A simple applicative language: Mini-ML*, Proceedings of the 1986 ACM Conference on Lisp and Functional Programming, ACM, August 1986, pp. 13–27.

[13] Roy Crole, *Hybrid adequacy*, Tech. Report CS-06-011, School of Mathematics and Computer Sience, University of Leicester, UK, November 2006.

[14] Haskell B. Curry and Robert Feys, *Combinatory logic*, vol. 1, North-Holland, Amsterdam, 1958.

[15] 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** (1972), 381–392.

[16] Joëlle Despeyroux, Amy Felty, and André Hirschowitz, *Higher-order abstract syntax in Coq*, Typed Lambda Calculi and Applications (Mariangiola Dezani-Ciancaglini and Gordon Plotkin, eds.), Lecture Notes in Computer Science, vol. 902, Springer, 1995, pp. 124–138.

[17] Amy Felty and Dale Miller, *Encoding a dependent-type λ-calculus in a logic programming language*, 10th International Conference on Automated Deduction (Mark Stickel, ed.), Lecture Notes in Computer Science, vol. 449, Springer, 1990, pp. 221–235.

[18] Amy Felty and Alberto Momigliano, *Web appendix of the paper "Hybrid: a definitional two level approach to reasoning with higher order abstract syntax" [21]*, `http://hybrid.dsi.unimi.it/jar/index.html`, May 2010, accessed September 2010.

[19] Amy P. Felty, *Two-level meta-reasoning in Coq*, Theorem Proving in Higher Order Logics (Victor Carreño, César Muñoz, and Sofiène Tahar, eds.), Lecture Notes in Computer Science, vol. 2410, Springer, 2002, pp. 198–213.

[20] Amy P. Felty and Alberto Momigliano, *Reasoning with hypothetical judgments and open terms in Hybrid*, Principles and Practice of Declarative Programming (António Porto and Francisco Javier López-Fraguas, eds.), ACM, 2009, pp. 83–92.

[21] Amy P. Felty and Alberto Momigliano, *Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax*, Journal of Automated Reasoning (2010), to appear.

[22] Marcelo P. Fiore, Gordon D. Plotkin, and Daniele Turi, *Abstract syntax and variable binding*, Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society, 1999, pp. 193–202.

[23] Murdoch Gabbay and Andrew M. Pitts, *A new approach to abstract syntax with variable binding*, Formal Aspects of Computing **13** (2002), no. 3–5, 341–363.

[24] Andrew Gacek, *The Abella interactive theorem prover (system description)*, Automated Reasoning (Alessandro Armando, Peter Baumgartner, and Gilles

Dowek, eds.), Lecture Notes in Computer Science, vol. 5195, Springer, 2008, pp. 154–161.

[25] Andrew Gacek, Dale Miller, and Gopalan Nadathur, *Combining generic judgments with recursive definitions*, Proceedings of the 23rd Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society, 2008, pp. 33–44.

[26] Jean-Yves Girard, *Linear logic*, Theoretical Computer Science **50** (1987), no. 1, 1–102.

[27] Georges Gonthier, *A computer-checked proof of the four colour theorem*, `http://research.microsoft.com/en-us/people/gonthier/4colproof.pdf`, April 2005, accessed August 2010.

[28] Georges Gonthier, *Formal proof—the four-color theorem*, Notices of the American Mathematical Society **55** (2008), 1382–1393.

[29] Andrew Gordon, *A mechanisation of name-carrying syntax up to α-conversion*, Higher Order Logic Theorem Proving and Its Applications (Jeffrey Joyce and Carl-Johan Seger, eds.), Lecture Notes in Computer Science, vol. 780, Springer, 1994, pp. 413–425.

[30] M. J. C. Gordon and A. M. Pitts, *The HOL logic and system*, Towards Verified Systems (J. Bowen, ed.), Real-Time Safety Critical Systems, vol. 2, Elsevier Science B.V., 1994, pp. 49–70.

[31] Mike Gordon, *From LCF to HOL: a short history*, Proof, Language, and Interaction: Essays in honour of Robin Milner, The MIT Press, Cambridge, Massachusetts, 2000, pp. 169–185.

[32] Robert Harper, Furio Honsell, and Gordon Plotkin, *A framework for defining logics*, Journal of the Association for Computing Machinery **40** (1993), no. 1, 143–184.

[33] Leon Henkin, *Completeness in the theory of types*, Journal of Symbolic Logic **15** (1950), no. 2, 81–91.

[34] Martin Hofmann, *Semantical analysis of higher-order abstract syntax*, Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society, 1999, pp. 204–213.

[35] HOL Team, *HOL4 Kananaskis 5*, `http://hol.sourceforge.net/`, October 2009, accessed August 2010.

[36] Douglas J. Howe, *Higher-order abstract syntax in classical higher-order logic*, LFMTP '09: Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages, ACM, 2009, pp. 1–11.

[37] Douglas J. Howe, *Higher-order abstract syntax in Isabelle/HOL*, Interactive Theorem Proving (Matt Kaufmann and Lawrence C. Paulson, eds.), Lecture Notes in Computer Science, vol. 6172, Springer, 2010, pp. 481–484.

[38] Brian Huffman, *Countable ordinals*, `http://afp.sourceforge.net/entries/Ordinal.shtml`, April 2009, accessed December 2009.

[39] Hybrid Group, *Hybrid: A package for higher-order syntax in Isabelle and Coq*, `http://hybrid.dsi.unimi.it/`, September 2010.

[40] Isabelle Team, *Isabelle*, `http://isabelle.in.tum.de/`, December 2009, accessed January 2010.

[41] Isabelle Team, *Isabelle 2008*, `http://isabelle.in.tum.de/website-Isabelle2008/index.html`, June 2008, accessed January 2010.

[42] Alexander Krauss and Andreas Schropp, *A mechanized translation from higher-order logic to set theory*, Interactive Theorem Proving (Matt Kaufmann and

Lawrence C. Paulson, eds.), Lecture Notes in Computer Science, vol. 6172, Springer, 2010, pp. 323–338.

[43] Donald MacKenzie, *Mechanizing proof: Computing, risk, and trust*, The MIT Press, 2001.

[44] Andy Magid (ed.), *Notices of the American Mathematical Society*, vol. 55, no. 11, December 2008, special issue on formal proof.

[45] Alan J. Martin, `http://hybrid.dsi.unimi.it/martinPhD/`, September 2010, Isabelle/HOL theory files.

[46] Raymond McDowell and Dale Miller, *Reasoning with higher-order abstract syntax in a logical framework*, ACM Transactions on Computational Logic **3** (2002), no. 1, 80–136.

[47] Raymond Charles McDowell, *Reasoning in a logic with definitions and induction*, Ph.D. thesis, University of Pennsylvania, 1997.

[48] Thomas F. Melham, *A mechanized theory of the π-calculus in HOL*, Nordic Journal of Computing **1** (1994), no. 1, 50–76.

[49] Alberto Momigliano and Simon Ambler, *Multi-level meta-reasoning with higher-order abstract syntax*, Foundations of Software Science and Computation Structures (Andrew Gordon, ed.), Lecture Notes in Computer Science, vol. 2620, Springer, 2003, pp. 375–391.

[50] Alberto Momigliano, Simon Ambler, and Roy L. Crole, *A Hybrid encoding of Howe's method for establishing congruence of bisimilarity*, Electronic Notes in Theoretical Computer Science **70** (2002), no. 2, 60–75.

[51] Alberto Momigliano, Alan J. Martin, and Amy P. Felty, *Two-level Hybrid: A system for reasoning using higher-order abstract syntax*, Electronic Notes in Theoretical Computer Science **196** (2008), 85–93.

[52] Alberto Momigliano and Jeff Polakow, *A formalization of an ordered logical framework in Hybrid with applications to continuation machines*, Proceedings of the 2003 ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with Variable Binding, MERLIN '03, ACM, 2003, pp. 1–9.

[53] Alberto Momigliano and Alwen Tiu, *Induction and co-induction in sequent calculus*, Types for Proofs and Programs – TYPES 2003 (Stefano Berardi, Mario Coppo, and Ferruccio Damiani, eds.), Lecture Notes in Computer Science, vol. 3085, Springer, 2004, pp. 293–308.

[54] George Necula, *Proof-carrying code*, 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, 1997, pp. 106–119.

[55] Tobias Nipkow, *Winskel is (almost) right: Towards a mechanized semantics*, Formal Aspects of Computing **10** (1998), no. 2, 171–186.

[56] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel, *Isabelle/HOL: A proof assistant for higher-order logic*, Lecture Notes in Computer Science, vol. 2283, Springer, 2002.

[57] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel, *Isabelle's logics: HOL*, `http://isabelle.in.tum.de/doc/logics-HOL.pdf`, December 2009, accessed January 2010.

[58] Frank Pfenning, *Logical frameworks*, Handbook of Automated Reasoning (Alan Robinson and Andrei Voronkov, eds.), Elsevier Science Publishers, 1999.

[59] Frank Pfenning and Conal Elliott, *Higher-order abstract syntax*, Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, 1988, pp. 199–208.

[60] Frank Pfenning and Carsten Schürmann, *System description: Twelf — A meta-logical framework for deductive systems*, Automated Deduction — CADE-16 (Harald Ganzinger, ed.), Lecture Notes in Computer Science, vol. 1632, Springer, 1999, pp. 202–206.

[61] Brigitte Pientka and Joshua Dunfield, *Programming with proofs and explicit contexts*, ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08), ACM Press, July 2008, pp. 163–173.

[62] Benjamin C. Pierce, *Types and programming languages*, The MIT Press, Cambridge, Massachusetts, 2002.

[63] Andrew Pitts, *The HOL logic*, Introduction to HOL: A theorem proving environment for Higher Order Logic (M. J. C. Gordon and T. F. Melham, eds.), Cambridge University Press, New York, NY, USA, 1993, pp. 191–232.

[64] Andrew M. Pitts, *Nominal logic, a first order theory of names and binding*, Information and Computation **186** (2003), no. 2, 165–193.

[65] Jeff Polakow, *Ordered linear logic and applications*, Ph.D. thesis, Carnegie Mellon University, 2001.

[66] Jeff Polakow and Frank Pfenning, *Relating natural deduction and sequent calculus for intuitionistic non-commutative linear logic*, 15th Conference on Mathematical Foundations of Programming Semantics (Andre Scedrov and Achim Jung, eds.), Electronic Notes in Theoretical Computer Science, vol. 20, Elsevier, 1999, pp. 449–466.

[67] R. Pollack and A. Momigliano, `mlLang.thy`, 2005, personal communication.

[68] Robert Pollack, *How to believe a machine-checked proof*, Twenty-Five Years of Constructive Type Theory (G. Sambin and J. Smith, eds.), Oxford University Press, 1998, pp. 205–220.

[69] N. Robertson, D. Sanders, P. Seymour, and R. Thomas, *The four-colour theorem*, Journal of Combinatorial Theory, Series B **70** (1997), 2–44.

[70] P. Rudnicki, *An overview of the Mizar project*, Proceedings of the 1992 Workshop on Types for Proofs and Programs (Bastad), Chalmers University of Technology, 1992.

[71] Robert J. Simmons, *Twelf as a unified framework for language formalization and implementation*, Tech. report, Princeton University, 2005, Undergraduate Senior Thesis 18679.

[72] Robert J. Simmons, *Mutable state*, `http://twelf.plparty.org/wiki/Mutable_state`, April 2008, accessed January 2010.

[73] Christian Urban, *Nominal techniques in Isabelle/HOL*, Journal of Automated Reasoning **40** (2008), no. 4, 327–356.

[74] Makarius Wenzel, *Isabelle/Isar — a generic framework for human-readable proof documents*, From Insight to Proof — Festschrift in Honour of Andrzej Trybulec (Roman Matuszewski and Anna Zalewska, eds.), University of Białystok, 2007.

[75] Markus Wenzel, *Type classes and overloading in higher-order logic*, Theorem Proving in Higher Order Logics (Elsa Gunter and Amy Felty, eds.), Lecture Notes in Computer Science, vol. 1275, Springer, 1997, pp. 307–322.

[76] Markus Wenzel and Freek Wiedijk, *A comparison of Mizar and Isar*, Journal of Automated Reasoning **29** (2002), 389–411.

[77] Alfred North Whitehead and Bertrand Russell, *Principia mathematica*, 3 vols., Cambridge University Press, 1910, 1912, 1913, second edition: 1925 (vol. 1), 1927 (vols. 2 & 3).

[78] Richard Zach, *Hilbert's program then and now*, Philosophy of Logic (Dale Jacquette, ed.), Handbook of the Philosophy of Science, vol. 5, Elsevier, Amsterdam, 2006, pp. 411–447.

Appendix A

Isabelle/HOL Formal Theories

The Isabelle/HOL theory corresponding to Section 3.3 (`HybridV.thy`) is given here for reference. It is an experimental and somewhat unpolished formal theory, although the proofs are mostly proper Isar.

The other formal theories presented in this thesis are available online [45].

HybridV.thy

```
(* A generalization of Hybrid to n-ary functions, by Alan J. Martin. *)

theory HybridV imports Hybrid begin (* Isabelle 2008 *)

declare to_expr_inject [iff] expr_ABS'_LAM [simp del]

types
  ind = nat
  'a iexp = "ind => 'a expr"
  'a ndB  = "'a iexp => 'a dB"
  'a nexp = "'a iexp => 'a expr"

abbreviation (input) INDn' :: "ind => 'a ndB"
  -- {* This is only an (input) abbreviation because
        it matches in too many unintended places when printing. *}
  where "INDn' i == (%v. dB (v i))"
abbreviation CONn' :: "'a => 'a ndB"
  where "CONn' a == (%v. CON' a)"
abbreviation VARn' :: "var => 'a ndB"
```

```
  where "VARn' n == (%v. VAR' n)"
abbreviation APPn' :: "['a ndB, 'a ndB] => 'a ndB"
  where "APPn' S T == (%v. S v $$' T v)"
abbreviation ERRn' :: "'a ndB"
  where "ERRn' == (%v. ERR')"
abbreviation BNDn' :: "bnd => 'a ndB"
  where "BNDn' j == (%v. BND' j)"
abbreviation ABSn' :: "'a ndB => 'a ndB"
  where "ABSn' S == (%v. ABS' (S v))"

abbreviation (input) INDn :: "ind => 'a nexp"
  -- {* This is only an (input) abbreviation because
        it matches in too many unintended places when printing. *}
  where "INDn i == (%v. v i)"
abbreviation CONn :: "'a => 'a nexp"
  where "CONn a == (%v. CON a)"
abbreviation VARn :: "var => 'a nexp"
  where "VARn n == (%v. VAR n)"
abbreviation APPn :: "['a nexp, 'a nexp] => 'a nexp"
  where "APPn S T == (%v. S v $$ T v)"
abbreviation ERRn :: "'a nexp"
  where "ERRn == (%v. ERR)"
abbreviation LAMn2 :: "[ind, 'a nexp] => 'a nexp"
  -- {* An alternate version of LAMn; it should work, but
        proving the necessary properties would be more difficult. *}
  where "LAMn2 i S == (%v. LAM x. S (v (i := x)))"

lemma dB_n:
  shows dB_INDn: "dB o (INDn i)   = INDn' i"
  and   dB_CONn: "dB o (CONn a)   = CONn' a"
  and   dB_VARn: "dB o (VARn n)   = VARn' n"
  and   dB_APPn: "dB o (APPn S T) = APPn' (dB o S) (dB o T)"
  -- {* no dB_LAMn, yet... *}
  and   dB_ERRn: "dB o ERRn       = ERRn'"
  unfolding o_def by simp_all

lemma expr_n':
  shows expr_INDn': "expr o (INDn' i)   = INDn i"
  and   expr_CONn': "expr o (CONn' a)   = CONn a"
  and   expr_VARn': "expr o (VARn' n)   = VARn n"
  and   expr_APPn': "[| Level 0 S; Level 0 T |] ==>
                      expr o (APPn' S T) = APPn (expr o S) (expr o T)"
  and   expr_ERRn': "expr o ERRn'       = ERRn"
  -- {* no expr_BNDn' (result is undefined) *}
  -- {* no expr_ABSn', yet... *}
  by (simp_all add: expand_fun_eq expr_transpose)

text {* Simplifier rules specific to 'a ndB. *}
```

```
lemma INDn'_neq_const [iff]: "INDn' i ~= (% v. s)"
  by (cases s)
     (auto simp add: expand_fun_eq intro: exI [of _ "%v. expr ERR'"]
                                          exI [of _ "%v. expr (CON' a)"])
lemmas const_neq_INDn' [iff] = INDn'_neq_const [symmetric]
lemma [simp]:
  shows INDn'_neq_APP':   "INDn' i ~= APPn' S T"
  and   INDn'_neq_ABS':   "INDn' i ~= ABSn' S"
  and   APP'_neq_INDn': "APPn' S T ~= INDn' i"
  and   ABS'_neq_INDn':   "ABSn' S ~= INDn' i"
  by (auto simp add: expand_fun_eq intro: exI [of _ "%v. expr ERR'"]
                                          exI [of _ "%v. expr (CON' a)"])

lemma nordinary_INDn' [iff]: "~ordinary (INDn' i)"
  by (simp add: ordinary_def)

text {* n-ary generalization of "abstr". *}

function Abstr_n :: "'a ndB => bool"
  where "Abstr_n (CONn' a)   = True"
  |     "Abstr_n (VARn' n)   = True"
  |     "Abstr_n (APPn' S T) = (Abstr_n S & Abstr_n T)"
  |     "Abstr_n (ERRn')     = True"
  |     "Abstr_n (BNDn' j)   = True"
  |     "Abstr_n (ABSn' S)   = Abstr_n S"
  |     "~ordinary S ==> Abstr_n S = (? i. S = INDn' i)"
  unfolding ordinary_def by atomize_elim auto
termination by (relation "measure (%S. size (S arbitrary))") auto

lemmas Abstr_n_cases = Abstr_n.cases [case_names CON VAR APP ERR BND ABS BASE]
  and Abstr_n_induct = Abstr_n.induct [case_names CON VAR APP ERR BND ABS BASE]

definition abstr_n :: "'a nexp => bool"
  where "abstr_n S = Abstr_n (dB o S)"

text {* Simplify abstr_n, except for LAMn (yet to be defined). *}

lemma Abstr_n_const: "Abstr_n (%v. s)"
  by (induct s) simp_all

lemma abstr_n_simps_part1 [iff]:
  shows abstr_n_INDn:  "abstr_n (INDn i)"
  and   abstr_n_const: "abstr_n (%v. s)"
  and   abstr_n_APPn:  "[| abstr_n S; abstr_n T |] ==> abstr_n (APPn S T)"
  by (auto simp add: abstr_n_def o_def Abstr_n_const)

text {* n-ary generalization of LAM *}

function Lbind_n :: "[ind, bnd, 'a ndB] => 'a ndB"
```

```
  where "Lbind_n i j (CONn' a)   = CONn' a"
  |     "Lbind_n i j (VARn' n)   = VARn' n"
  |     "Lbind_n i j (APPn' S T) = APPn' (Lbind_n i j S) (Lbind_n i j T)"
  |     "Lbind_n i j ERRn'       = ERRn'"
  |     "Lbind_n i j (BNDn' k)   = BNDn' k"
  |     "Lbind_n i j (ABSn' S)   = ABSn' (Lbind_n i (Suc j) S)"
  |     "~ordinary S ==> Lbind_n i j S = (if S = INDn' i then BNDn' j else S)"
  unfolding ordinary_def by atomize_elim auto
termination by (relation "measure (%(i,j,S). size (S arbitrary))") auto

definition Lambda_n :: "[ind, 'a ndB] => 'a ndB"
  where "Lambda_n i S = (if Abstr_n S then ABSn' (Lbind_n i 0 S) else ERRn')"

definition LAMn :: "[ind, 'a nexp] => 'a nexp"
  where [unfolded Lambda_n_def]: "LAMn i S = expr o (Lambda_n i (dB o S))"

lemma Abstr_n_Lbind_n [simp]: "Abstr_n S ==> Abstr_n (Lbind_n i j S)"
  by (induct S arbitrary: j rule: Abstr_n_induct) simp_all

lemma Level_Lbind_n [simp]: "Level j S ==> Level (Suc j) (Lbind_n i j S)"
  by (induct S arbitrary: j rule: Abstr_n_induct) simp_all

lemma dB_LAMn:
  "dB o LAMn i S = (if abstr_n S
                    then ABSn' (Lbind_n i 0 (dB o S))
                    else ERRn')"
  unfolding LAMn_def abstr_n_def o_def by simp

lemma abstr_n_LAMn [simp]: "abstr_n S ==> abstr_n (LAMn i S)"
  by (simp add: abstr_n_def LAMn_def o_def)

text {* Injectivity of LAMn *}

lemma dB_o_inject: "dB o S = dB o T ==> S = T"
  by (simp add: expand_fun_eq dB_inject)

lemma Lbind_n_inject:
  assumes "Abstr_n S" and "Abstr_n S'" and "Level j S" and "Level j S'"
  shows "Lbind_n i j S = Lbind_n i j S' <-> S = S'"
using assms proof (induct S arbitrary: j S' rule: Abstr_n_induct)
  apply_end (case_tac [!] S' rule: Abstr_n_cases)
qed auto

lemma LAMn_inject_E:
  assumes "LAMn i S = LAMn i T" "abstr_n S | abstr_n T"
  obtains "S = T"
proof -
  { fix S T :: "'a nexp"
    assume p: "LAMn i S = LAMn i T" "abstr_n S"
```

```
    have "S = T"
    proof (cases "abstr_n T")
      case True -- {* abstr_n T *}
      let ?S = "dB o S" and ?T = "dB o T" note o_def [simp]
      from ‘abstr_n S‘ and ‘abstr_n T‘
      have aS: "Abstr_n ?S" and aT: "Abstr_n ?T"
        unfolding abstr_n_def by simp_all
      with ‘LAMn i S = LAMn i T‘ [unfolded LAMn_def abstr_n_def]
      have "expr o (ABSn' (Lbind_n i 0 ?S)) = expr o (ABSn' (Lbind_n i 0 ?T))"
        by simp
      with aS aT have "?S = ?T"
        by (simp add: expand_fun_eq Lbind_n_inject [unfolded expand_fun_eq])
      thus "S = T" by (simp add: expand_fun_eq dB_inject)
    next
      case False with p show ?thesis unfolding LAMn_def and abstr_n_def
        by (simp add: o_def expand_fun_eq expr_transpose)
    qed }
  note r = this
  show ?thesis using assms by (auto intro: that r sym)
qed

text {* Datatype-like distinctness and injectivity lemmas for type nexp. *}

lemma nexp_distinct_E_part1:
  "CONn a = VARn n ==> P"
  "CONn a = APPn S T ==> P"
  "CONn a = ERRn ==> P"
  "VARn n = CONn a ==> P"
  "VARn n = APPn S T ==> P"
  "VARn n = ERRn ==> P"
  "APPn S T = CONn a ==> P"
  "APPn S T = VARn n ==> P"
  "APPn S T = ERRn ==> P"
  "ERRn = CONn a ==> P"
  "ERRn = VARn n ==> P"
  "ERRn = APPn S T ==> P"
  by (simp_all add: expand_fun_eq)

lemma nexp_distinct_E_part2:
  "CONn a = INDn i ==> P"
  "VARn n = INDn i ==> P"
  "APPn S T = INDn i ==> P"
  "ERRn = INDn i ==> P"
  "INDn i = CONn a ==> P"
  "INDn i = VARn n ==> P"
  "INDn i = APPn S T ==> P"
  "INDn i = ERRn ==> P"
  by (auto simp add: expand_fun_eq)
```

```
lemma nexp_distinct_E_part3:
  "INDn i = LAMn j S ==> P"
  "CONn a = LAMn j S ==> P"
  "VARn n = LAMn j S ==> P"
  "APPn S T = LAMn j S ==> P"
  "[| ERRn = LAMn j S; ~P ==> abstr_n S |] ==> P"
  "LAMn j S = INDn i ==> P"
  "LAMn j S = CONn a ==> P"
  "LAMn j S = VARn n ==> P"
  "LAMn j S = APPn S T ==> P"
  "[| LAMn j S = ERRn; ~P ==> abstr_n S |] ==> P"
  by (auto simp add: expand_fun_eq dB_inject [symmetric]
                     dB_LAMn [unfolded o_def, THEN fun_cong]
           split: if_splits elim: allE [where x = "%i. VAR 0"])

lemmas nexp_distinct_E [elim!] =
  nexp_distinct_E_part1 nexp_distinct_E_part2 nexp_distinct_E_part3

lemma nexp_inject_E_part1:
  "[| CONn a1 = CONn a2; a1 = a2 ==> P |] ==> P"
  "[| VARn n1 = VARn n2; n1 = n2 ==> P |] ==> P"
  "[| APPn S1 T1 = APPn S2 T2; [| S1 = S2; T1 = T2 |] ==> P |] ==> P"
  by (simp_all add: expand_fun_eq)

lemma nexp_inject_E_part2:
  "[| INDn i1 = INDn i2; i1 = i2 ==> P |] ==> P"
  by (auto simp add: expand_fun_eq elim: allE [where x = "%i. VAR i"])

lemmas nexp_inject_E [elim!] =
  nexp_inject_E_part1 nexp_inject_E_part2 LAMn_inject_E

lemma nexp_distinct [simp]:
  "                INDn i ~= CONn a"
  "                INDn i ~= VARn n"
  "                INDn i ~= APPn S T"
  "                INDn i ~= LAMn j S"
  "                INDn i ~= ERRn"
  "                CONn a ~= INDn i"
  "                CONn a ~= VARn n"
  "                CONn a ~= APPn S T"
  "                CONn a ~= LAMn j S"
  "                CONn a ~= ERRn"
  "                VARn n ~= INDn i"
  "                VARn n ~= CONn a"
  "                VARn n ~= APPn S T"
  "                VARn n ~= LAMn j S"
  "                VARn n ~= ERRn"
  "              APPn S T ~= INDn i"
  "              APPn S T ~= CONn a"
```

```
  "               APPn S T ~= VARn n"
  "               APPn S T ~= LAMn j S"
  "               APPn S T ~= ERRn"
  "               LAMn j S ~= INDn i"
  "               LAMn j S ~= CONn a"
  "               LAMn j S ~= VARn n"
  "               LAMn j S ~= APPn S T"
  "abstr_n S ==> LAMn j S ~= ERRn"
  "                   ERRn ~= INDn i"
  "                   ERRn ~= CONn a"
  "                   ERRn ~= VARn n"
  "                   ERRn ~= APPn S T"
  "abstr_n S ==>      ERRn ~= LAMn j S"
  by auto

lemma nexp_inject [simp]:
  "                     (INDn i1 = INDn i2) <-> (i1 = i2)"
  "                     (CONn a1 = CONn a2) <-> (a1 = a2)"
  "                     (VARn n1 = VARn n2) <-> (n1 = n2)"
  "               (APPn S1 T1 = APPn S2 T2) <-> (S1 = S2 & T1 = T2)"
  "abstr_n S ==>      (LAMn i S = LAMn i T) <-> (S = T)"
  "abstr_n T ==>      (LAMn i S = LAMn i T) <-> (S = T)"
  by auto

text {* Size induction support for type nexp *}

definition "size_n (S :: 'a nexp) = size (S (%i. ERR))"
definition "size_n' (S :: 'a ndB) = size (S (%i. ERR))"

lemma size_Lbind_n:
  "Abstr_n S ==> size_n' S < Suc (size_n' (Lbind_n j k S))"
  unfolding less_Suc_eq_le and size_n'_def
proof (induct S arbitrary: k rule: Abstr_n_induct)
  apply_end (erule_tac [3] x = k in meta_allE)
  apply_end (erule_tac [3] x = k in meta_allE)
qed auto

lemma Level_dB_o: "Level i (dB o S)"
  unfolding o_def by simp

lemma size_n_monos [simp]:
  "size_n (INDn i) = 0"
  "size_n (CONn a) = 0"
  "size_n (VARn n) = 0"
  "size_n (APPn S T) = Suc (size_n S + size_n T)"
  "abstr_n S ==> size_n S < size_n (LAMn j S)"
  "size_n ERRn     = 0"
proof -
  show "abstr_n S ==> size_n S < size_n (LAMn j S)"
```

```
    by (simp add: size_n_def LAMn_def abstr_n_def o_def expr_size_def
                  size_Lbind_n [unfolded size_n'_def, of "%x. dB (S x)"])
qed (simp_all add: size_n_def)

text {* Structural induction for type nexp *}

function Linv_n :: "[ind, bnd, 'a ndB] => 'a ndB"
  where "Linv_n i j (CONn' a)   = CONn' a"
  |     "Linv_n i j (VARn' n)   = VARn' n"
  |     "Linv_n i j (APPn' S T) = APPn' (Linv_n i j S) (Linv_n i j T)"
  |     "Linv_n i j ERRn'       = ERRn'"
  |     "Linv_n i j (BNDn' k)   = (if k = j then INDn' i else BNDn' k)"
  |     "Linv_n i j (ABSn' S)   = ABSn' (Linv_n i (Suc j) S)"
  |     "~ordinary S ==> Linv_n i j S = S"
  unfolding ordinary_def by atomize_elim auto
termination by (relation "measure (%(i,j,S). size (S arbitrary))") auto

text {*
  I would like to say that Lbind_n and Linv_n are inverses.
  However, there will be conditions on both sides: Linv_inverse will require
  choice of a fresh index, while Lbind_inverse will have a level condition.
  (Which is essentially a freshness condition, and I'm considering changing
  the treatment of fresh indices to something more conventional.)
*}

lemma Lbind_n_inverse:
  assumes "Level j S" shows "Linv_n i j (Lbind_n i j S) = S"
  using assms by (induct S arbitrary: j rule: Abstr_n_induct) simp_all

text {*
  I *really* need a better convention for the various "equivalent" predicates
  on terms, functions, etc., and for the corresponding theorems.

  I could certainly overload the predicates, but it may be unwise to do so in
  the absence of an axiomatic scheme allowing the theorems to be overloaded
  as well.  I'll try it for the freshness predicates and see how it works.

  Observations so far:
  - The need for a separate name in definitions is unfortunate, but tolerable.
  - Type inference may assign overly general types to the arguments of
    overloaded predicates.  (This happened in fresh_to_func below.)
*}

consts fresh :: "[ind, 'a] => bool"

overloading Fresh_n == fresh begin
function Fresh_n :: "[ind, 'a ndB] => bool"
  where "     Fresh_n i (CONn' a) <-> True"
  |     "     Fresh_n i (VARn' n) <-> True"
```

```
  |     "   Fresh_n i (APPn' S T) <-> Fresh_n i S & Fresh_n i T"
  |     "         Fresh_n i ERRn' <-> True"
  |     "     Fresh_n i (BNDn' j) <-> True"
  |     "     Fresh_n i (ABSn' S) <-> Fresh_n i S"
  |     "~ordinary S ==> Fresh_n i S <-> S ~= INDn' i"
  unfolding ordinary_def by atomize_elim auto
termination by (relation "measure (%(i,S). size (S arbitrary))") auto
end

lemma INDn'_inject [iff]: "(INDn' i1 = INDn' i2) <-> (i1 = i2)"
  by (auto simp add: expand_fun_eq elim: allE [where x = VAR])

lemma dB_not_INDn' [iff]: "~(! t v. dB t = INDn' i v)"
  by (auto intro!: exI [where x = "VAR 0"] exI [where x = "%j. VAR 1"])

lemma Fresh_n_INDn' [iff]: "fresh i (INDn' j) <-> j ~= i"
  by (auto simp add: expand_fun_eq intro!: exI [where x = "%j. VAR j"])

lemma Fresh_n_subst:
  "Abstr_n S ==> fresh i S <-> (! t v. S (v (i := t)) = S v)"
proof (induct S rule: Abstr_n_induct)
  case (BASE S) then obtain j where "S = INDn' j" by auto
  thus ?case
    by (auto simp add: expand_fun_eq intro!: exI [where x = "%j. VAR j"])
qed auto

lemma Abstr_n_fresh_cofinite:
  assumes "Abstr_n S" shows "finite {i. ~fresh i S}"
  using assms by (induct S rule: Abstr_n_induct)
                 (auto simp del: disj_not1
                       simp add: finite_Un [unfolded Un_def mem_def],
                  simp_all add: Collect_def)

lemma Abstr_n_ex_fresh:
  assumes "Abstr_n S"
  obtains i where "fresh i S"
proof -
  have "? i. i ~: {j. ~fresh j S}"
  proof (rule ex_new_if_finite)
    show "~finite (UNIV :: nat set)" by (rule infinite_UNIV_nat)
    show "finite {i. ~fresh i S}" using `Abstr_n S`
      by (rule Abstr_n_fresh_cofinite)
  qed
  with that show thesis by auto
qed

lemma Linv_n_inverse:
  assumes "fresh i S" shows "Lbind_n i j (Linv_n i j S) = S"
  using assms by (induct S arbitrary: j rule: Abstr_n_induct) auto
```

```
lemma Linv_n_0_cases [consumes 1]:
  "[| fresh i S; S = Lbind_n i 0 (Linv_n i 0 S) ==> P |] ==> P"
  by (simp add: Linv_n_inverse)

lemma Level_Linv_n [simp]:
  "[| Abstr_n S; Level (Suc j) S |] ==> Level j (Linv_n i j S)"
  by (induct S arbitrary: j rule: Abstr_n_induct) auto

lemma Abstr_n_Linv_n [simp]: "Abstr_n S ==> Abstr_n (Linv_n i j S)"
  by (induct S arbitrary: j rule: Abstr_n_induct) auto

lemma expr_o_inverse: "Level 0 S ==> dB o (expr o S) = S"
  unfolding o_def expand_fun_eq by auto

lemma expr_ABSn'_LAMn_l1:
  assumes "Abstr_n S" and "Level 0 S"
  shows "expr o (ABSn' (Lbind_n i 0 S)) = LAMn i (expr o S)"
proof -
  from ‘Level 0 S‘ and ‘Abstr_n S‘ have "abstr_n (expr o S)"
    by (simp add: abstr_n_def expr_o_inverse)
  thus ?thesis using ‘Level 0 S‘ unfolding LAMn_def abstr_n_def
    by (simp add: expr_o_inverse)
qed

lemma expr_ABSn'_LAMn:
  assumes "fresh i S" and "Abstr_n S" and "Level (Suc 0) S"
  shows "expr o (ABSn' S) = LAMn i (expr o (Linv_n i 0 S))"
using ‘fresh i S‘ proof (cases rule: Linv_n_0_cases)
  assume S_expand: "S = Lbind_n i 0 (Linv_n i 0 S)"
  have "expr o (ABSn' (Lbind_n i 0 (Linv_n i 0 S)))
        = LAMn i (expr o (Linv_n i 0 S))"
    using ‘Abstr_n S‘ and ‘Level (Suc 0) S‘
    by (simp add: expr_ABSn'_LAMn_l1)
  thus "expr o (ABSn' S) = LAMn i (expr o (Linv_n i 0 S))"
    using S_expand [symmetric] by simp
qed

lemma nexp_nchotomy_l1:
  assumes "Abstr_n S" and "Level (Suc 0) S"
  shows "? i T. abstr_n T & ABSn' S = dB o (LAMn i T)"
proof -
  obtain i where "fresh i S" using ‘Abstr_n S‘ by (rule Abstr_n_ex_fresh) auto
  let ?T = "expr o (Linv_n i 0 S)"
  from ‘fresh i S‘ and ‘Abstr_n S‘ have a: "LAMn i ?T = expr o (ABSn' S)"
    by (rule expr_ABSn'_LAMn [symmetric])
  from ‘Abstr_n S‘ and ‘Level (Suc 0) S‘
  have l0: "Level 0 (Linv_n i 0 S)" and l0a: "Level 0 (ABSn' S)" by simp_all
  from a and ‘Abstr_n S‘ have "abstr_n ?T"
```

```
    by (simp add: abstr_n_def expr_o_inverse [OF l0])
  moreover from a have "ABSn' S = dB o (LAMn i ?T)"
    by (simp add: expr_o_inverse [OF l0a])
  ultimately show ?thesis by auto
qed

lemma nexp_nchotomy_l2:
  assumes "Abstr_n Y" and "Level 0 Y"
  shows "(? i. Y = (INDn' i)) | (? a. Y = CONn' a) | (? n. Y = VARn' n)
       | (? S T. Abstr_n S & Abstr_n T
              & Level 0 S & Level 0 T & Y = APPn' S T)
       | (? i T. abstr_n T & Y = dB o (LAMn i T)) | Y = ERRn'"
proof (cases Y rule: Abstr_n_cases)
  case (ABS S) with assms show ?thesis
    using nexp_nchotomy_l1 [where S = S] by simp
qed (insert assms, simp_all)

lemma dB_expr_o_transpose: "Level 0 T ==>  dB o S = T  <->  S = expr o T"
  unfolding o_def by auto

lemma six_cases_E:
  "[| A' | B' | C' | D' | E' | F';
      A' ==> A; B' ==> B; C' ==> C; D' ==> D; E' ==> E; F' ==> F |]
   ==> A | B | C | D | E | F" by auto

lemma dB_o_inverse: "expr o dB = id"
  by (simp add: expand_fun_eq)

lemma nexp_nchotomy:
  assumes "abstr_n Y"
  shows "(? i. Y = INDn i) | (? a. Y = CONn a) | (? n. Y = VARn n)
       | (? S T. Y = APPn S T) | (? i S. abstr_n S & Y = LAMn i S) | Y = ERRn"
proof -
  from`abstr_n Y` have "Abstr_n (dB o Y)"
    unfolding abstr_n_def .
  moreover have "Level 0 (dB o Y)" by auto
  ultimately have
    "(? i. (dB o Y) = INDn' i) | (? a. (dB o Y) = CONn' a)
   | (? n. (dB o Y) = VARn' n)
   | (? S T. Abstr_n S & Abstr_n T
          & Level 0 S & Level 0 T & (dB o Y) = APPn' S T)
   | (? i T. abstr_n T & (dB o Y) = (dB o LAMn i T))
   | (dB o Y) = ERRn'" by (rule nexp_nchotomy_l2)
  thus ?thesis
    by (rule six_cases_E)
       (auto simp add: dB_expr_o_transpose expr_n' o_assoc LAMn_def
                       dB_o_inverse)
qed
```

```
lemma abstr_n_APPn_simp [simp]:
  "abstr_n (APPn S T) <-> abstr_n S & abstr_n T"
  by (simp add: abstr_n_def o_def)

lemma nexp_exhaust:
  assumes abstr_n: "abstr_n Y"
  obtains (INDn)   i where "Y = INDn i"
  |       (CONn)   a where "Y = CONn a"
  |       (VARn)   n where "Y = VARn n"
  |       (APPn) S T where "Y = APPn S T" and "abstr_n S" and "abstr_n T"
  |       (LAMn) i S where "Y = LAMn i S" and "abstr_n S"
  |       (ERRn)           "Y = ERRn"
  using nexp_nchotomy [where Y = Y] and abstr_n
  by auto auto -- {* why? *}

lemma nexp_induct [consumes 1, case_names INDn CONn VARn APPn LAMn ERRn]:
  assumes "abstr_n U"
  and "!! i. P (INDn i)"
  and "!! a. P (CONn a)"
  and "!! n. P (VARn n)"
  and "!! S T. [| P S; P T |] ==> P (APPn S T)"
  and "!! i S. P S ==> P (LAMn i S)"
  and "P ERRn"
  shows "P U"
using ‘abstr_n U‘ proof (induct U rule: measure_induct [of size_n])
  case (1 U) from ‘abstr_n U‘ show ?case
  proof (cases rule: nexp_exhaust)
    case (APPn S T)
    hence "P S" and "P T"
      by (auto intro!: "1.hyps" [unfolded ‘U = APPn S T‘, rule_format])
    thus "P U" unfolding ‘U = APPn S T‘ by (rule assms)
  next
    case (LAMn i S)
    hence "P S"
      by (auto intro!: "1.hyps" [unfolded ‘U = LAMn i S‘, rule_format])
    thus "P U" unfolding ‘U = LAMn i S‘ by (rule assms)
  qed (insert assms, simp_all)
qed

declare to_expr_inject [iff del] expr_ABS'_LAM [simp]

end
```