AN APPROACH TO COMPILER CORRECTNESS USING INTERPRETATION BETWEEN THEORIES

Beth Helene Levy

December 1986
CSD-860032

UNIVERSITY OF CALIFORNIA

Los Angeles

An Approach to Compiler Correctness
Using Interpretation Between Theories

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Computer Science

by

Beth Helene Levy

1986

© Copyright by

Beth Helene Levy

1986

The dissertation of Beth Helene Levy is approved.

Kirby A. Baker

Daniel M. Berry

Stephen D. Crocker

Sheila A. Greibach

Yiannis N. Moschovakis

David F. Martin, Committee Chair

University of California, Los Angeles

1986

Table of Contents

List of Figures

ACKNOWLEDGEMENTS

First and foremost, I would like to thank Prof. David Martin for introducing me to semantics and correctness proofs, for inspiring this dissertation, and for having the perseverance to advise and encourage me throughout this research. I would like to thank Dr. Stephen Crocker for several years of encouragement and support. Both David Martin's and Stephen Crocker's belief that verification is a worthwhile and viable technology has influenced my work. Professors Daniel Berry, Sheila Greibach, Yiannis Moschavakis, and Kirby Baker, as members of my Ph.D committee, have been very responsive. The classes that I took from David Martin, Daniel Berry, and Sheila Greibach were always challenging and inspiring, and influenced this research.

I would like to thank Mitchell Wand, John Reynolds, Joseph Goguen, Tim Redmond, and Peter Dybjer. They generously provided helpful comments and responded to requests for information. My friends and colleagues Anne Brindle, Steve Kelem, and Mary Vernon came through for me during some difficult times and I cannot thank them enough. Carol LeDoux helped me modify Scribe databases in order to format the dissertation.

The Aerospace Corp. and Hughes Aircraft Co. provided economic support and computing resources during my research. I would not have been able to complete my studies without their educational support. Kathy Nauman at The Aerospace Corp. did an outstanding job in helping me type the dissertation.

Last, but certainly not least, I would like to thank Lloyd Bookman for many years of unwavering support, encouragement, and faith in me. My research would have been impossible without this.

VITA

	Born, Los Angeles, California
1972	University of California, Los Angeles Alumni Scholarship
1972-77	University of California, Los Angeles Dean's Honor List, University of California, Los Angeles Honor's Program, University of California, Los Angeles Departmental Scholar, Phi Beta Kappa
1977	B.S., University of California, Los Angeles
1977	University of California, Los Angeles Mathematics-Computer Science Senior Prize
1973-1982	Member of the Technical Staff, Hughes Aircraft Company El Segundo, California
1974-1982	Recipient of Hughes Aircraft Company B.S., M.S., and Ph.D. Fellowships
1979	M.S., University of California, Los Angeles
1982-present	Member of the Technical Staff, The Aerospace Corporation El Segundo, California

ABSTRACT OF THE DISSERTATION

An Approach to Compiler Correctness
Using Interpretation Between Theories

by

Beth Helene Levy
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 1986
Professor David F. Martin, Chair

An approach to compiler correctness verification is proposed and research that investigates the mathematical framework and applicability of the approach is presented. This approach is based on interpretation between theories, a concept developed in mathematical logic which provides a basis for proving that one logical theory is correctly mapped into another logical theory. To utilize this concept for the compiler application, it is proposed that the theories include higher-order operators (operators that accept operators as arguments and/or return operators as results) and domain equations. Interpretation between theories has previously been defined for predicate calculus and DLP (an extension of dynamic logic).

An extension to predicate calculus is proposed which incorporates Scott's theory of domains. It allows higher-order operators and recursive objects, and can be used to specify the denotational semantics of a programming language. An interpretation

between our extended theories and criteria the interpretation must meet to be correct are defined. In the course of developing the definitions, we prove various theorems that show the criteria are sufficient. The interpretation between theories can be used as a formal specification of a compiler design. A mathematical proof that the interpretation is correct constitutes a verification that the compiler design is correct.

The novel concepts presented by this approach are:

1. Interpretation between theories is defined for theories that allow higher-order operators and domain equations
2. A compiler design is defined as an interpretation between theories.

Preliminary research indicates that this approach has strong intuitive appeal because it models the informal design process, results in concise specifications, and organizes the correctness proof into highly modularized, manageable pieces.

Chapter 1

Introduction

There are several reasons why the application of analytical software verification techniques to the validation of compilers[1] is important to software/hardware development. First, in many installations compilers are heavily used and typically have long useful lives. Thus, errors in compilers will likely be more costly than errors in less-used programs. Second, an error in a compiler is often difficult to distinguish from an error in the source language input to the compiler. Programmers using a compiler should not have to become familiar with the internal workings of the compiler or with the target code produced (or integrated circuits produced in the case of "silicon compilers") to distinguish compiler errors from programming errors. A third and perhaps more subtle consideration is that any proof that a program written in a source language is correct is useless if an error exists in the compiler used to translate that program. Finally, by focusing our attention on the compiler correctness problem we will hopefully contribute to the solutions of other difficult verification problems in software and hardware applications.

It is useful to focus on the compiler problem because the problem is well understood and motivated, and highly structured. Like the verification of other

[1]A compiler is a computer program that translates a program written in a language which cannot be directly executed (called the source language) into another language which can be directly executed (called the target language).

classes of software, compiler verification involves showing that a computer program correctly implements a specification. A compiler specification (or compiler design) is a functional statement of what the compiler is supposed to do; it is the relation between the source program input and the target program output. A compiler implementation is a computer program that implements the compiler design. Contrary to some other applications, it is also important to verify the design; the compiler input and output are computer programs and the semantics of any valid source program input must be shown to be preserved by the target program output. This means we must have (1) a description of the semantics of both the source and target languages in addition to the semantics of the programming language in which the compiler is written, and (2) another layer of proof (i.e., a proof that the compiler design is correct).

A proof that a compiler design is correct requires a semantic definition method. The area of semantics is not as well developed as the area of syntax specification. Backus-Naur form (BNF) or context-free grammars are now widely used for defining syntax and constructing parsers[2]. When first introduced, BNF was considered too difficult to learn and cumbersome to use to be of practical use. Much of the same criticism can be heard of various semantics definition methods today. In spite of the early criticism, formal syntactic methods are now taught to and used by beginning programmers, and the methods have had a profound effect on the design of programming languages and compilers.

Currently, there is no standard method for writing semantics. As mentioned

[2]a process for reading and verifying the proper syntax of input

above, the area is not as well developed as syntax specification; semantics features are much more difficult to define and describe [schmidt 86]. Semantic definition methods are evolving in response to the various needs of language implementers and programmers. These include [schmidt 86]:

1. A precise standard for a computer implementation. The semantics of a source language is independent of any particular computer or compiler. Different compilers can implement the same source language for different machines. Such a source language is said to be portable. The purpose of a standard is to guarantee that the source language is implemented in exactly the same manner on all machines.
2. User documentation. Just as a trained programmer can read a formal syntax definition, a semantic definition can be used as a reference to answer subtle questions about the behavior or interaction of programming language constructs.
3. A tool for design and analysis. Analogous to syntax definitions which can be used as input to parser generator systems, semantic definitions can be input to compiler generator systems or used to suggest efficient, elegant implementations. They can also be used for testing and analyzing a language. These areas of research are still evolving.

All of this suggests that while the selection of a semantic definition method is important, there is no one clear choice for all purposes; different definition methods were developed in response to different, sometimes competing, goals. In this research, a denotational style semantic definition method was selected. It is an expressive and convenient method for semantic definitions. The motivation for this choice is discussed in detail in the dissertation. Briefly, the goals motivating the selection are:

1. A concise, unambigious semantics specification
2. A method that has been demonstrated with a wide variety of languages
3. A definition method that could be incorporated into the proposed verification approach which has as goals a correctness proof based on structural induction on the source language, a correctness proof that mirrors the informal process of design/verification, and a correctness proof that breaks down into small, independent, manageable pieces.

At this point some general comments on the problem of verification might be useful. There is no such thing as absolute correctness; correctness is a relative term. An implementation is correct with respect to some specification, design, or requirements. A proof of correctness demonstrates that properties of the specification are preserved by the implementation. If the language of the specification does not easily convey the specification's intent, the specification can in turn be defined in another language or in a more abstract (less detailed) manner, and these two specifications can be shown consistent, *ad infinitum*. Because one person's specification may be another person's implementation in the hierarchy of design, the distinction between an implementation and a specification blurs, and the distinction is only helpful when viewing two levels of design. When considering two levels of a design, it is desirable that the languages of the implementations and their specifications, and the proofs of correctness be as formal as possible, leaving little room for misinterpretation.

A proof in a formal system is a precise, convincing argument that an implementation is consistent with its specification. Without such a formal proof, one cannot communicate, document, or reproduce verification results in a uniform manner. The more rigorous and formal the proof techniques are, the less confusion there is about the validity of the results and the more amenable the process is to mechanization, i.e., computer assistance. However, the intent is not to put a straightjacket on creativity nor mask intuition with formalisms. The proofs involved are not particularly difficult, though some are quite long. Many of the proofs are similar in nature. Mechanization eliminates much of the tedium involved and reduces the chance of error in a proof. It allows us to tackle larger verification problems and forces us to be precise.

A formal correctness proof increases one's confidence in an implementation, but does not guarantee correctness -- it is likely that nothing can. Where can the verification process break down? First, the proof could be wrong or unsound. Next, the specification could be wrong; it might not accurately convey one's mental concept. So, the question arises whether the additional cost of formally verifying an implementation or design is worthwhile. We believe it is for certain classes of software.

Consider the alternatives to correctness proofs. Testing or code walkthroughs are currently used to certify the correctness of most software. For critical software, however, it is not adequate to trust a system on the basis that the code or design has been examined for some sample input. That sample input is necessarily a small percentage of the total range allowable. It is the nature of programming that one can write an executable program without having completely understood the problem. Hence, it is typical that systems are unreliable for the first few months or years of operation. For compilers, it means that hundreds or thousands of programmers encounter costly delays, may themselves produce erroneous programs via an erroneous compiler, or write code that will avoid errors in the compiler, code that is hard to understand and maintain.

Testing and code walkthroughs are based on informal or ambiguous specifications. Intuitively, the correctness proof can be considered a formal, systematic code walkthrough in which the full range of possible input is consciously examined. It provides a permanent written record of this reasoning. The mere act of formal specification slows down the implementation process, forcing the designer to carefully consider error conditions, bounds of a range, structuring of code and data, etc. At the very least, for critical software, certification must include something more than testing. The sophistication of our software systems is surpassing our means to certify them, to have confidence in them. Correctness proofs are not intended to replace the other certification techniques, but rather, to augment them.

It is recognized that the cost of formal verification is very great, and in most instances, given the current state of technology, the cost is too great to absorb. It is evident that correctness proofs will never be used to certify "one-shot", short-lived programs. The long-lived systems where costs (labor, and equipment) and risks (e.g., human life, national defense, loss of money) are great, have the need for improved reliability and can absorb more initial development costs. Systems such as compilers, operating systems, networks, microcode, etc. are indispensible parts of long-lived environments. They are heavily used on a daily basis and must be correct. More verification research is required. Perhaps verification methods for particular types of applications must be developed and the tedious, more mundane parts of the verification process must be automated. Automation may include processes from checking the syntax of a specification or checking a proof to finding a proof and/or implementation.

The work presented in this dissertation was done with some of these problems and goals in mind. It is hoped that it will provide more insight into the nature of implementing a specification/requirement and will illustrate the large amount of reasoning that must be done formally, now done informally in someone's head, to justify such an implementation.

This dissertation proposes to apply the mathematical concept of interpretation between theories to the verification of non-optimizing compilers. A compiler is non-optimizing if the compilation of each syntactic type is independent of other syntactic types. The proof of a non-optimizing compiler can be conveniently divided into two parts. The first part proves that the compiler design is correct; i.e., the target language which is output by the compiler preserves the semantics of the source language which is input to the compiler. The second part proves that the compiler implementation is correct; i.e., for each source language syntactic type the compiler produces a particular sequence of target instructions. In this dissertation we apply the concept of interpretation between logical theories to the first part of the correctness proof. Further research will determine whether this formulation of the problem and proof method can be used in the second part of the correctness proof. The presentation of the research completed is outlined below.

It is important to note that this report uses several words that have different meanings in different contexts or references. These words include semantics, syntax, interpretation, structure and implementation. In this dissertation:

1. semantics refers to the meaning/behavior given to programming languages
2. syntax refers to the grammatical structure of programming languages or of a theory's language
3. structure or model refers to the meaning given to a theory (in other papers this is often referred to as the interpretation or semantics).
4. interpretation refers to the mapping from the language of one theory into the language of another theory.
5. an implementation is formally specified by an interpretation; for the compiler problem, it can refer to either a compiler design or an interpretation from the compiler design to a programming language (this is referred to as the compiler implementation above).

In [wand 80], it is postulated that a programming language is just a complex abstract data type where an abstract data type is a set of operations and the definitions of the relationships between the operations. The evaluation of a program is another operation in the data type. The evaluation operation is more commonly referred to as the interpreter or operational semantics of the language. The evaluation operation may be formulated in terms of homomorphisms (denotational or algebraic semantics). In this dissertation, we also specify programming languages as abstract data types where the programming language semantics are operations in the data type. The rules that show how to evaluate or simplify the semantics are included in the data type.

A systematic, organized specification of an abstract data type is given by a "logical theory." A theory for an abstract data type consists of a language for the data type and a statement of the properties of the data type on which reliance can be placed. An interpretation between theories is a mapping that defines how one theory (data type) is implemented by another theory (data type).

In [wand 82a], Wand extends the concept of interpretation between theories from predicate calculus to dynamic logic. The extension includes interpretations of procedures, equality, and tuples of sorts. This extended definition of interpretation between theories can be applied to the correctness problem of abstract data types that commonly occur in computer applications, e.g., stacks. For background and reference material, refer to Appendix A for a description of interpretation between first-order theories. Refer to Appendix B and Chapter 3 for a description of Wand's criteria for correct implementation of abstract data types in terms of interpretation between theories.

Chapter 2 discusses the application of the approach to the compiler verification problem. In particular, we discuss why it is desirable to define an implementation as an interpretation between theories.

Chapters 4 and 5 provide the background and motivation for this dissertation's proposed extension to interpretations which is presented in Chapter 6. The approach is extended to accommodate higher order abstract data types (many-sorted theories with function space types), and thus, it can be applied to complicated abstract data types such as programming languages. The application of the approach to compiler design correctness is described and demonstrated with examples in Chapters 7 and 8. In Chapter 9 this approach is compared to other methods that have been applied to the compiler problem, including the algebraic method. Finally, Chapter 10 proposes future work.

The contributions of this project are that:

1. the application of interpretation between theories to the compiler correctness problem will be investigated
2. interpretation between theories will be extended to include theories that have higher order operators and domains, and that
3. the foundation will be laid for a verification system.

The goals of this project are:

1. to define a verification method that models the informal process of changing a representation and then determining whether the change of representation is correct, and
2. to define a verification method that is highly modular so that many items in the verification task can be done in parallel and possibly mechanized, and minor changes to specifications will have little effect on any existing verification.

Chapter 2

Application of Interpretation Between Theories to the Compiler Correctness Problem

A broad overview of the verification approach is presented in this chapter. As mentioned in the Introduction, the proof of a non-optimizing compiler can be divided into two parts. The first part proves that the target language which is output by the compiler preserves the semantics of the source language which is input to the compiler [chirica 86]. Call this proof the *compiler design correctness proof*. This proof is necessary because there can be more than one correct output for some particular input to the compiler, the input can be arbitrarily large, and the preservation of the source language's semantics in the output is not obvious. The second part of the compiler correctness proof proves that for each source language syntactic type the compiler produces a particular sequence of target instructions [chirica 86]. Call this proof the *compiler implementation correctness proof*.

The proposed approach for the compiler design correctness proof is to define the source and target languages as abstract data types. Each abstract data type is specified as a logical theory. Call the theory for the source language T_{source}. The language, axioms, and rules of inference of T_{source} are denoted L_{source}, A_{source} and R_{source}, respectively (similarly for the target language). The nonlogical symbols in L_{source} are the names of syntactic constructs and semantic operators of the source language that are implemented in L_{target}. Thus, a compiler design is an interpretation of the nonlogical symbols of T_{source} and equality into the language of the implementing theory T_{target}. The axioms and rules specify the programming language properties on which one relies (i.e., they specify the programming language semantics). The interpretation is extended to formulas, and thus, can be used to translate the axioms and rules. The interpreted axioms specify the implementation of the source programming language semantics. In Chapter 6 languages for the theories are discussed and examples are presented in Chapter 8.

Assuming T_{source} and T_{target} are sound, a necessary condition for the compiler design to be correct is that the interpretation of the axioms and rules in T_{source} be deducible in T_{target}. This means the semantics of the source language are preserved in its implementation. This is one of several correctness criteria. How do we know the list is complete -- that the criteria will ensure a correct implementation?

In [wand 82a], Wand defines the properties of a correct interpretation in his Implementation Theorem. An interpretation is correct if the interpretation of any source theorem is a target theorem and if a structure for the source theory can be constructed from a target theory structure. In other words, an interpretation is correct if the implementation of any deducible source property is deducible in the implementing environment and if the source behavior can be perceived in the behavior of the implementation. In terms of the compiler specification, a compiler specification is correct if any true property about any source program is true in the implementation and if the behavior of any source program can be perceived in the behavior of the compiled source program.

By using interpretation between theories as a formal description of implementation, we have a sound mathematical basis for the concept of correct implementation without requiring that an implementation be defined as a model or as a homomorphism. See Appendix B for a detailed discussion. The key point is that a domain of source objects can be implemented as some subset of a domain of target objects and a source object can have many equivalent representations in the implementation. The correctness proof is nicely structured into a translation process and a deduction which only uses T_{target}. Furthermore, since we assume the only L_{source} formulas are the ones generated by the axioms and rules in T_{source}, we can easily determine how *any* source language phrase is implemented and be assured that this implementation is correct. The key characteristics of the compiler design correctness proof are:

1. the proof proceeds by structural induction on the source language;
2. the structural induction argument is implicity handled by using the interpretation to translate axioms;
3. the implementation of both source programming language syntax and semantics is treated in a uniform manner;
4. the correctness proof does not require knowledge of a theory's structure; it can be done at the syntactic level of the formal system;
5. the correctness proof utilizes only the target theory.

Although the compiler implementation correctness problem is not addressed in this paper, the goal is to develop a method that enables one to use part of the compiler design as the specification in the compiler implementation correctness proof. One way to think about this is to extract that part of the compiler design that deals with the implementation of syntactic constructs in the source language and ignore the implementation of source language semantics. Then the compiler implementation is an interpretation from that part of the compiler design concerned with source syntax to a theory for the programming language in which the compiler is written. The specification describes the input/output relation the compiler program must satisfy. This is amenable to a Floyd/Hoare verification approach where a program is proved consistent with an input/output specification, with a precondition and a postcondition, because it appears that the input/output relations are first-order expressions and the implementation of source and target syntactic domains should be relatively straightforward [chirica 86].

Chapter 3

Interpretation Between Theories for a Many-Sorted Predicate Calculus

3.1. Correctness Proofs Based on Interpretation Between Theories

In [wand 82a] Wand is concerned with the specification and correct implementation of abstract data types (e.g., stacks). A specification of an abstract data type is a set of formulas in some logical language; it is a theory. In [wand 82a] the logical language is DLP (Dynamic Logic of Programs). The operators of the data type are nonlogical symbols of DLP and appear in the formulas. The formulas are formal statements of the properties of the abstract data type and are true or false given a particular structure for the language of the data type.

The implementation of an abstract data type is defined as an interpretation (mapping or translation) of the language of the theory for the abstract data type into another theory's language. Wand based his definition of implementation on an extension of interpretation between theories from predicate calculus to DLP. The extension allows interpretation of procedure symbols, interpretation of sorts as tuples, and interpretation of equality symbols. The extension requires that free variables in the interpreted formulas be restricted to those values that are "legal" implementations of the variables' sorts. A formula is introduced to decide whether a value is a legal representation.

Wand defines the criteria which any correct implementation must satisfy. He proves that if the criteria are met then the "reasonable" properties one expects of a correct implementation, which he specifies in the Implementation Theorem, are satisfied.

In the following sections we will simplify Wand's results. We eliminate procedure symbols and concentrate on dealing with the interpretation of equality and sort symbols. Wand presents his results in a semantic or model-theoretic manner. We briefly review that approach in this chapter. However, in extending Wand's work we provide the axiomatics and give proof-theoretic versions of the results.

3.2. Syntax and Structure of a Specification Language

The nonlogical symbols of a first-order language are the quantifier symbol, predicate symbols, function symbols, and constant symbols. DLP, as described by Wand, extends a first-order language by adding sort symbols and procedure symbols. We will simplify Wand's discussion by eliminating procedure symbols.[3] It follows that the simplified language discussed here is a many-sorted first-order language. All operator symbols have a signature, and terms and formulas are constructed in the usual way (see [wand 82a], [enderton 72], or Appendix A for the grammar and other details).

The semantics of the specification language is given by a *structure* that assigns "meanings" to the set of nonlogical symbols. The meanings are extended to apply to terms and formulas. The structure M is given as a function on each language symbol as follows:

[3]At this time we do not anticipate the need for procedure symbols in compiler design correctness proofs. However, we may reconsider this when the approach is applied to compiler implementation correctness proofs.

1. sort symbol: for each sort symbol σ, $M(\sigma) = U_\sigma$, where U_σ is a nonempty set. U_σ is called the carrier of sort σ. U denotes the union of the sets U_σ as σ ranges over the sort symbols.
2. function symbol: for each function symbol $f: \sigma_1 \times \ldots \times \sigma_n \rightarrow \sigma$, M assigns a function $f^M: U_{\sigma_1} \times \ldots \times U_{\sigma_n} \rightarrow U_\sigma$.
3. predicate symbol: for each predicate symbol $p: \sigma_1 \times \ldots \times \sigma_n$, M assigns a predicate p^M on $U_{\sigma_1} \times \ldots \times U_{\sigma_n}$, such that for the distinguished symbol $=_\sigma$, M assigns the equality predicate. Because we eliminate procedure symbols in this discussion, predicate symbols are treated as function symbols with the distinguished codomain bool (bool stands for boolean values).

A state ρ is a function from the set of individual variable symbols to U. A state is sort preserving in the sense that if v is an individual variable symbol of sort σ, then $\rho(v) \in U_\sigma$.

M is extended to terms by mapping a term to a function where the function maps states to carriers (i.e., M: terms $\rightarrow$ states $\rightarrow$ U). Specifically,

1. if x is an individual variable symbol then $M(x)(\rho) = \rho(x)$
2. if $t_1, \ldots, t_n$ are terms of sorts $\sigma_1, \ldots, \sigma_n$ and f is an n-place function symbol with signature $\sigma_1 \times \ldots \times \sigma_n \rightarrow \sigma$, then $M(ft_1 \ldots t_n)(\rho) = f^M(M(t_1)(\rho), \ldots, M(t_n)(\rho))$.

M is extended to formulas by mapping a formula to a function where the function maps states to boolean values (i.e., M: formulas $\rightarrow$ states $\rightarrow$ bool). Specifically, if G and H range over formulas then

1. if $pt_1 \ldots t_n$ is an atomic formula then $M(pt_1 \ldots t_n)(\rho) = p^M(M(t_1)(\rho), \ldots, M(t_n)(\rho))$.
2. $M(G \,\&\, H)(\rho) = M(G)(\rho) \,\&\, M(H)(\rho)$
3. $M(G \vee H)(\rho) = M(G)(\rho) \vee M(H)(\rho)$
4. $M(\neg G)(\rho) = \neg M(G)(\rho)$
5. $M(G \supset H)(\rho) = M(\neg G)(\rho) \vee M(H)(\rho)$
6. $M((\forall_D v)F)(\rho) = M(F)(\rho')$, for all ρ' such that $\rho=\rho'$ except possibly at v (i.e., $\rho v \neq \rho' v$)

3.3. Implementation Defined as an Interpretation

The implementation of an abstract data type is defined as an interpretation of the language of the theory for the abstract data type (e.g., language of stacks) into another theory's language (e.g., language of array-integer pairs). If L_1 and L_2 are many-sorted first-order languages of theories T_1 and T_2, respectively, then an interpretation I of L_1 in L_2 is an assignment of phrases of L_2 to each nonlogical symbol of L_1 as follows:

1. to each sort symbol σ of L_1 assign a sort symbol σ^I of L_2 and for each sort symbol σ create a formula is-σ with signature $\sigma^I \rightarrow$ bool.
2. to each function symbol f: $\sigma_1 \times \ldots \times \sigma_n \rightarrow \sigma$ assign a term f^I in L_2 in which variables $v_1, \ldots, v_n$ occur free and for $1 \leq i \leq n$, v_i: σ_i.[4]
3. to each predicate symbol p: $\sigma_1 \times \ldots \times \sigma_n \rightarrow$ bool assign a formula p^I in L_2 in which variables $v_1, \ldots ,v_n$ occur free and for $1 \leq i \leq n$, v_i: σ_i.
4. to each individual variable symbol v with signature σ assign an individual variable symbol v^I in L_2 with signature σ^I.

To extend the interpretation to terms and formulas, we must first define free variables, and preambles of formulas. If α is a well-formed formula it has a set FV(α) of free variables. The set is defined inductively by:

1. FV(x) = {x}, where x is a variable
2. FV(ht_1 ... t_n) = FV(t_1) $\cup \ldots \cup$ FV(t_n) where h is an n-place function or predicate symbol
3. FV(($\forall$v)α) = FV(α) - {v}

The preamble of α is a formula pre(α) and is defined by:

pre(α) = is-D_1(I(x_1)) & . . . & is-D_n(I(x_n))

where FV(α) = {$x_1, \ldots , x_n$} and for $1 \leq i \leq n$, x_i: D_i.

[4]This differs from [wand 82a] but agrees with [enderton 72]. Furthermore, the notation v_i: σ_i means that variable v_i has signature σ_i.

The interpretation of formula α is $(pre(\alpha) \supset I(\alpha))$ where I is extended as follows:

1. $I(ht_1 \ . \ . \ . \ t_n) = (I(h)I(t_1) \ . \ . \ . \ I(t_n))$, where h is an n-place function or predicate symbol.
2. $I((\forall_D v)\ F) = ((\forall_{I(D)} I(v))(is\text{-}D(I(v)) \supset I(F)))$
3. $I(G \text{ op } H) = (I(G) \text{ op } I(H))$, where op $\in \{\&, \vee, \supset\}$
4. $I(\neg\ G) = (\neg\ I(G))$

The interpretation is not a structure because equality is interpreted as any equivalence relation and free variables in the interpreted axiom are restricted. The interpretation cannot be used to define a homomorphism from one model to another, but rather, a homomorphism to a partitioned subset of a model.[5]

3.4. Correctness Criteria

Let T_1 be a theory in language L_1 and T_2 be a theory in language L_2. A correct implementation of T_1 in T_2 is an interpretation I of L_1 in L_2 such that the following formulas are a logical consequence of T_2:

1. $(\exists x)(is\text{-}\sigma(x))$ for each sort σ of L_1
2. $is\text{-}\sigma_1(x_1)\& \ldots \& \ is\text{-}\sigma_n(x_n) \supset is\text{-}\sigma(f^I x_1 \ldots x_n)$ for each function symbol f with signature $\sigma_1 \times \ldots \times \ \sigma_n \rightarrow \sigma$ in L_1
3. $I(x =_\sigma x)$ for each sort σ of L_1
4. $I(x_1 = y_1 \ \& \ldots \& \ x_n = y_n \supset fx_1 \ \ldots \ \ x_n = fy_1 \ldots y_n)$ for each n-place function symbol f.
5. $I(x_1 = y_1 \ \& \ \ldots \ \& \ x_n = y_n \supset (px_1 \ \ldots \ \ x_n \supset py_1 \ \ldots \ y_n))$ for each n-place predicate symbol p
6. I(F) for each axiom F of T_1 (i.e., $T_1 \vdash F$)

Condition 1 states that the carrier of the interpretation of each sort is nonempty. Condition 2 is required because sorts have been introduced into the language. It states that if the input data satisfy the formula (invariant) of their sort, then the

[5]In algebraic terminology, the interpretation maps to a quotient of the subalgebra of the implementing algebra. The operators of the subalgebra are contained in the operators of the implementing algebra and all operations of the subalgebra are appropriate restrictions of those for the implementing algebra.

output of the interpreted function satisfies the formula (invariant) of its sort. Conditions 3 and 5 are necessary because equality must be interpreted as an equivalence relation. They state the interpretation of equality is a reflexive relation and is preserved by the interpretation of predicates. Condition 4 states that functions are preserved in the interpretation. Condition 6 states that the translation of the axioms of T_1 are logical consequences of T_2; the properties of the data type on which one relies are preserved in its implementation.

3.5. The Implementation Theorem

If the six correctness conditions are satisfied then the following theorem, called the *Implementation Theorem*, is true: Let I be a correct implementation of T_1 in T_2. Then,

1. if A_2 is any L_2-structure, there is an L_1-structure A_1 such that for any closed formula F of L_1, $\models_{A_1} F$ if and only if $\models_{A_2} I(F)$.
2. for any formula F of L_1, if $T_1 \vdash F$ then $T_2 \vdash I(F)$.

This section discusses Wand's proof of the Implementation Theorem. This proof justifies the existence of the correctness criteria. It will provide an outline for reproving the Implementation Theorem for other theories and guide the construction of correctness criteria.

The Implementation Theorem defines the properties of a correct implementation. The first part of the theorem gives the "synthetic view" of implementation. It states, given a model A_2 of T_2, that we should be able to construct a model A_1 of T_1; given the behavior of the implementing objects (e.g., behavior of an array and a pointer) we should be able to perceive the implemented object (e.g., behavior of a stack). The second part of the theorem gives the "analytic view" of implementation. It states "if we reason about the implemented theory T_1, we should be able to draw conclusions about the implementation." For example, if we deduce that a stack does not underflow and a stack is implemented by an array and a pointer, then we should be able to predict that the pointer has a particular lower bound. These two parts of the theorem should hold for any "reasonable notion of specification language and correct implementation."

Wand's notion of specification language and correct implementation supports the view that "the use of specifications as a tool for information hiding and of implementation as translation is a naturally occuring phenomenon. Consider a specification for a GCD (greatest common denominator) module. We implement the specification by writing a GCD program in PASCAL, which is translated by the PASCAL compiler into P-code, which is translated into machine code, which is translated by the digital architecture into actions of registers and busses . . . . Each such translation is typically called an 'implementation' of the preceding level. At every level the implementation forgets what is involved both above and below the translation." At every level the implementation should preserve the properties above in the implementing environment below. This is the informal meaning of the Implementation Theorem.

The proof of the Implementation Theorem is broken down into a set of proofs. Eliminating procedure symbols, the following are a list of theorems used to prove the Implementation Theorem:

1. **Theorem 3.1** If states ρ_1 and ρ_2 agree on FV(G), then $M(G)(\rho_1)$ iff $M(G)(\rho_2)$.
2. **Lemma 4.1.** If $x_1, \ldots, x_k$ include (perhaps properly) the free variables of G, and $T_2 \vdash \text{is-}\sigma_1(I(x_1))\ \&\ \ldots\ \&\ \text{is-}\sigma_k(I(x_k)) \supset I(G)$, then $T_2 \models \text{pre}(G) \supset I(G)$
3. **Lemma 4.2.** If I is an interpretation of T_1 in T_2 and t is a term of sort σ in L_1, then $T_2 \models \text{pre}(t) \supset \text{is-}\sigma(I(t))$.

4. **Lemma 4.5.** Let I be an interpretation of T_1 in T_2, let A_2 be any L_2-structure, and σ be any sort of L_1. Then the interpretation of $=_\sigma$ induces an equivalence relation on that subset $U_{I(\sigma)}$ where is-σ is true.
5. **Theorem 4.1** Let I be an interpretation of T_1 in T_2, and let A_2 be an L_2-structure. Then there is an L_1-structure A_1 and a map J from states of A_2 to states of A_1 such that for any formula G of L_1 and state ρ of A_2 such that M(pre(G))ρ=true, M(G)(Jρ) iff M(I(G))ρ.
6. **Corollary 4.1.** If G is a closed formula, then $A_1 \models G$ iff $A_2 \models I(G)$.
7. **Theorem 4.2** Let I be an interpretation of T_1 in T_2. If T_1 logically implies G, then T_2 logically implies I(G).

The core of the Implementation Theorem proof is the proof of Theorem 4.1. It is described in the next section. Theorem 4.1 deals with the construction of the implemented structure A_1 from the implementing structure A_2. Wand shows how to handle the interpretation of equality and restriction of variables in the interpreted formulas. The proofs of the other theorems are described in [wand 82a].

Wand also shows that a theory with tuples can be implemented in a first-order theory. A first-order theory is extended to include a set of operator symbols and axioms that specify tuples. In a manner similar to Wand we will extend first-order theories to include Scott's theory of domains. In addition to products we will add domain equations, sums, and function spaces. This "augmented" theory will be referred to as the theory schema or ancestor theory. Both the source and target theories will be constructed from the theory schema; both our implemented and implementing environment have higher order objects. We will not show that the theory schema can be implemented in a first-order theory.

3.5.1. Theorem 4.1

Theorem 4.1 states that given an L_2-structure A_2 and an interpretation I of T_1 in T_2, we can construct an L_1-structure A_1 and a map J from A_2-states to A_1-states such that the following holds. For any formula G of L_1 and A_2-state ρ, such that M(pre(G))ρ=true, we have M(G)(Jρ) iff M(I(G))ρ. This means that given a model for the implementing data type, a model for the implemented data type can be constructed.

3.5.1.1. Proof

In the proof, superscripts s and t are used in place of A_1 and A_2; s and t stand for the source (the specification or implemented object) and the target (the implementing object), respectively. $U^t_{I(\sigma)}$ denotes the carrier of sort I(σ) in A_2. Denote the is-σ subset of $U^t_{I(\sigma)}$ by $V^t_{I(\sigma)}$. By lemma 4.5 $I(=_\sigma)$ is an equivalence relation in the target theory. Let $=_\sigma$ denote the equivalence relation $I(=_\sigma)$ on $V^t_{I(\sigma)}$.

The L_1-structure A_1 is constructed from the L_2-structure A_2 as follows:

1. for each sort σ of L_1, let $U^s_\sigma = V^t_{I(\sigma)} /{=}$. This is nonempty by correctness condition 1 which states that $T_2 \models (\exists x)(\text{is-}\sigma(x))$.
2. for each function symbol f: $\sigma_1 \times \ldots \times \sigma_n \rightarrow \sigma$ of L_1, let f^s: $U_{\sigma_1} \times \ldots \times U_{\sigma_n} \rightarrow U_\sigma$: $([a_1],\ldots,[a_n]) \rightarrow [I(f)^t a_1 \ldots a_n]$ where square brackets denote equivalence classes. The definition is independent of the choice equivalence class representatives because by correctness condition 4 we have $T_2 \models I(x_1=y_1 \& \ldots \& x_n=y_n \supset fx_1 \ldots x_n = fy_1 \ldots y_n)$.
3. for each predicate symbol p: $\sigma_1 \times \ldots \times \sigma_n \rightarrow$ bool of L_1, let p^s: $U^s_{\sigma_1} \times \ldots \times U^s_{\sigma_n} \rightarrow U^s_{bool}$: $([a_1],\ldots,[a_n]) \rightarrow I(p)^t a_1 \ldots a_n$. The definition is independent of the choice of equivalence class representatives because by correctness condition 5 we have $T_2 \models I(x_1=y_1 \; \& \ldots \& \; x_n=y_n \supset (px_1 \ldots x_n \supset py_1 \ldots y_n))$.

Thus, U^s_σ is a partitioned subset of $U^t_{I(\sigma)}$. The subset is defined by is-σ and the partition by the interpretation of equality as an equivalence relation. There are target objects that do not represent source objects. Each source object can be represented by any one of several equivalent target objects. The derived source operations are restricted to operate on partitioned subsets of target objects.

Next, the map J from states of A_2 to states of A_1 is defined. For each sort σ of L_1, let e_σ be an arbitrarily chosen element of U^s_σ. Define J_σ: $U^t_{I(\sigma)} \to U^s_\sigma$ by $J_\sigma a = [a]$ if is-σ(a) and $J_\sigma a = e_\sigma$ otherwise. Define J: $(Var^t \to U^t) \to (Var^s \to U^s)$ as $J\rho v = J_\sigma(\rho I(v))$ where v has sort σ.

Now we have the definitions of I, M, and J. Assuming $M(pre(G))\rho$ = true, we proceed to show that $M(G)(J\rho)$ iff $M(I(G))\rho$ by structural induction on the formula G; we prove results for terms, atomic formulas, and then formulas.

We must first show that if t is a term and $M(pre(t))\rho$ = true then $M(t)(J\rho) = [M(I(t))\rho]$. By induction on terms we have

1. if t = x where x is a variable of sort σ and $\rho(I(x))$ = a then $M(x)(J\rho) = J\rho x = J_\sigma\rho(I(x)) = J_\sigma a = [a] = [\rho I(x)] = [M(I(x))\rho]$. Note that $J_\sigma a = [a]$ because by assumption $M(pre(t))\rho = M(is\text{-}\sigma(I(x))\rho$ = true.
2. if $t = ft_1...t_n$ where f is an n-place function symbol, $M(I(f))\rho = I(f)^t$, $M(I(t_1))\rho = a_1,..., M(I(t_n))\rho = a_n$, then $M(ft_1...t_n)(J\rho)$

   $= f^s M(t_1)(J\rho)...M(t_n)(J\rho)$, by definition of M
   $= f^s[a_1] ... [a_n]$, by induction hypothesis
   $= [I(f)^t a_1...a_n]$, by definition of f^s
   $= [M(I(ft_1...t_n))\rho]$, by definition of M

By lemma 4.2 and the assumption we have $M(ft_1 ... t_n)(J\rho) \in U^s_\sigma$.

Next consider atomic formulas. We must show that if $M(pre(pt_1...t_n))\rho$ = true then $M(pt_1...t_n)(J\rho)$ = true iff $M(I(pt_1...t_n))\rho$ = true.

$M(pt_1...t_n)(J\rho)$

$= p^s\ M(t_1)(J\rho)\ ...\ M(t_n)(J\rho)$

$= I(p)^t\ M(I(t_1))\rho\ ...\ M(I(t_n))\rho$, by definition of p^s and induction hypothesis

$= M(I(pt_1,...,t_n))\rho$

Assuming the results hold for terms and atomic formulas, it is easy to show the result for formulas. We will only consider the case where G is of the form $(\forall_\sigma v)F$. Let equiv(ρ, ρ′, x) mean that for all w not equal to x, ρ(w) = ρ′(w).

$M((\forall_\sigma v)F)(J\rho)$

= for all $(J\rho)'$ such that equiv($(J\rho)$, $(J\rho)'$,v), $M(F)(J\rho)'$

= for all $J\rho'$ such that equiv($J\rho$,$J\rho'$,v), $M(F)(J\rho')$

= for all ρ′ such that equiv(ρ,ρ′,I(v)), $M(I(F))\rho'$, by induction hypothesis

$= M((\forall_{I(\sigma)}v)\ I(F))\rho'$

$= M(I((\forall_\sigma v)F))\rho'$

Chapter 4

Domains in Denotational Semantics

One of the goals is to define a theory schema that allows one to specify the abstract syntax and semantics of a programming language. In particular, denotational semantics is the style of semantics that we selected. To specify the denotational semantics of a programming language the theory must allow domains, a class of "structured" sets; domain operators and domain equations are incorporated into the language for the theory schema and there are axioms for reasoning about domains. The basic operators for defining domains are $\otimes$ (product), $\oplus$ (sum), and $\rightarrow$ (function space). Domains can be defined recursively via domain equations.

Many features of programming languages can be given a denotational semantics without bringing in any mathematics other than sets and total functions over them. However, there are several problems for which it is necessary or convenient to use sets with structure [wadsworth 78]. These include

1. non-termination: the semantics of non-terminating computations must be specified. This is handled by allowing partial functions in which the results of some non-terminating computations may be undefined.
2. higher-order procedures: some programming languages allow higher-order procedures (arguments and/or results of a procedure may themselves be procedures). Higher-order functions may be used to specify the semantics. The semantics of a procedure is the function (argument-value pairs) it computes. The semantics of a procedure is expressed in terms of the semantics of the arguments and results. Hence, the use of higher-order functions. This is different from an operational (or interpreter) semantic description where the semantics of a procedure is represented by a structured object (sometimes called a closure) which contains among other things the text of the procedure body. In operational semantics textual information is operated on and passed to various functions.

3. recursive procedures: it may be necessary to define procedures recursively in programming languages and we would like to specify their semantics. If the functions involved map between the specially structured sets called domains, then the recursive definitions can be treated as equations for which there is guaranteed a solution. Fixed point methods can be used to solve the equations.

4. recursive data types: there are recursive data types (e.g., lists, trees) in programming languages. They can also arise in giving the semantics of a programming language even if the language does not allow recursive procedures. Analogous to recursive procedures, it is desirable to treat the recursive definitions as equations. If the data types are modelled as domains, then the equations are guaranteed a solution.

For the compiler correctness problem, denotational semantics is particularly useful. In the first place, the semantics for a wide variety of programming languages have been specified with denotational semantics: These include:

- ALGOL60 [henhapl 82, mosses 74],
- ALGOL68 [milne 72],
- Pascal [andrews 82, tennent 77],
- LISP [gordon 73, muchnick 82],
- SNOBOL [tennent 73],
- Ada [bjorner 80, donzeau-gouge 80, kini 82],
- Lucid [ashcroft 82],
- CHILL [branquart 82],
- Scheme [muchnick 82]

Next, a domain can be viewed as a data type for semantics. Its name plus its operations constitute what is referred to as a semantic algebra in [schmidt 86]. This is nicely integrated into our approach where a data type is also specified by a semantic algebra, which in this context is referred to as a theory, i.e., domain names, operators, and formulas that describe domain properties.

Finally, denotational semantics is useful for the compiler problem because the semantics for each syntactic construct is concisely and unambigously stated in a formula, independent of the semantics for the other syntactic constructs. Generally speaking, if the semantics or specification is changed for one syntactic construct, it can be done independently of the other constructs' semantics. Thus, for verification, a small perturbation in a specification will only effect a small change in a correctness proof. Specifications with higher order operators are concise and "abstract". All this makes for a structured correctness proof with short, independent pieces.

It is the purpose of this paper to show how domains are incorporated into the theory, define an interpretation between such theories, define correctness criteria for the interpretation, and prove that satisfaction of the correctness criteria will result in a "correct" or "reasonable" implementation.

Chapter 5

Scott's Theory of Domains

The goal is to extend a many-sorted first-order language to include a class of structured sets called domains. Dana Scott's theory of domains ensures that every recursive definition of a function or recursive definition of a set is "good"; it guarantees that all equations, possibly recursive, have a unique solution. Scott showed in the early 1970's how to define a model that will allow both kinds of recursive objects; there is a consistent theory for dealing with these recursive objects. Recursively defined sets are objects called domains and recursive functions are elements of particular domains called function spaces. These objects are important for semantic definitions of non-trivial programming languages.

New domains are constructed using various *domain constructors*. The constructors are $\otimes$ (product), $\oplus$ (sum), and $\rightarrow$ (function space). Others, such as * (finite sequences), can be defined in terms of these three constructors. Recursive domains are defined in domain equations. In the following chapters we will add the three constructors and domain equations to the basic theory schema. We will consider for each domain construction the operators, axioms, interpretation, and justification of the correctness criteria for the interpretation.

The structure we use assigns cpo's (complete partially ordered sets) to signatures generated from domain constructors and sort symbols. The remainder of this section will briefly review some key properties of cpo's. This was primarily extracted from [barendregt 81], [chirica 76], [mosses 75], and [schmidt 86].

Definition 1: A *preorder* is a reflexive and transitive relation.

Definition 2: A *partial order* is a preorder which is also antisymmetric. We use $\leq$ to denote partial order.

Definition 3: A *poset* (partial ordered set) P is a nonempty set together with a partial order on P.

Definition 4: An *upper bound* (ub) for $X \subseteq P$ is any element $p \in P$ such that $x \leq p$ for all $x \in X$. If for any other upper bound x' of X, $p \leq x'$ then p is said to be the *least upper bound* of X in P, denoted lub(X).

Definition 5: If the empty set has a lub in P it is called the *bottom element*, denoted $\perp_p$, or $\perp$ when there is no danger of confusion. $\perp_p$ has the property that $\perp_p \leq p$ for all $p \in P$.

For completeness' sake, directed sets, cartesian products, and separated sums are defined. However, in the course of justifying the proposed verification method, chains, coalesced products, and coalesced sums suffice. Also, we use complete partial orders - other work on domains has been based on lattices.

Definition 6: A nonempty subset $X \subseteq P$ is *directed* iff $(\forall x, y \in X)(\exists z \in X)(x \leq z \text{ and } y \leq z)$. In other words, $\text{lub}(\{x, y\}) \in X$.

Definition 7: A subset $X \subseteq P$ is a *chain* in P iff $(\forall x, y \in X)(x \leq y \text{ or } y \leq x)$.

Definition 8: P is a *complete partial order* (cpo) iff

1. There is a bottom element in P, and
2. Every directed subset $X \subseteq P$ has a lub or every nonempty chain has a lub (the latter referred to as chain complete).

This is also referred to as a pointed or strict cpo. N.B. a *domain* is a cpo.

Definition 9: Let P and Q be posets. A function f: $P \rightarrow Q$ is *monotonic* iff $(\forall p, p' \in P)(p \leq p' \text{ implies } f(p) \leq f(p'))$.

Definition 10: Let P and Q be posets. A function f: $P \to Q$ is *continuous* iff for all directed $X \subseteq P$, $f(\mathrm{lub}(X)) = \mathrm{lub}(f(X))$ where $f(X) = \{f(x) \mid x \in X\}$. If X is a nonempty chain then we have f is chain continuous.

Theorem 11: Continuous maps on cpo's are always monotonic.

Definition 12: A poset is *strict* iff it has a bottom element. If P and Q are strict posets then a function f: $P \to Q$ is called strict iff $f(\bot_p) = \bot_q$.

Definition 13: A continuous and bijective function f: $P \to Q$ between domains is called a *domain isomorphism*; in this case P and Q are called *isomorphic domains*, written P = Q. This is also called a *domain equation*. If P and Q are strict, then f must be a strict function.

Proposition 14: For any set A, the disjoint union $A_\bot = A \cup \{\bot\}$, with the partial order $a \le b$ iff $a = \bot$ or $a = b$, is a strict domain. Such domains are called *flat domains*.

Proposition 15: Given domains A and B, let $A \times B$ be the *cartesian product of domains* partially ordered by $<a,b> \le <a',b'>$ iff $a \le_A a'$ and $b \le_B b'$. Then $A \times B$ is a domain with for $X \subseteq A \times B$, $\mathrm{lub}(X) = <\mathrm{lub}(pr1(X)), \mathrm{lub}(pr2(X))>$ where the projection functions pr1: $A \times B \to A$ and pr2: $A \times B \to B$ are continuous and $pr1(X) = \{x \in A \mid (\exists x' \in B) <x,x'> \in A \times B\}$. Similarly for pr2(X). A cartesian product is a domain.

Proposition 16: The *coalesced product* of A and B, written $A \otimes B$, is defined as $\{<a,b> \in A \times B \mid a \ne \bot_A \text{ and } b \ne \bot_B\} \cup \{\bot\}$. It is partially ordered by $<a,b> \le <a',b'>$ iff $<a,b> = \bot$ or ($a \le_A a'$ and $b \le_B b'$). A coalesced product is a domain.

Proposition 17: The *separated sum* of A and B, written A + B, is defined as $\{<0, a> \mid a \in A\} \cup \{<1,b> \mid b \in B\} \cup \{\bot\}$. It is partially ordered by $x \le x'$ iff

1. $x = \bot$, or
2. $x = <0,x>$ and $x' = <0,x'>$ and $x \le_A x'$, or
3. $x = <1,x>$ and $x' = <1,x'>$ and $x \le_B x'$

A separated sum of domains is a domain.

Proposition 18: A *coalesced sum* of A and B, written $A \oplus B$, is defined as $\{<0,a> \mid a \in A \text{ and } a \ne \bot_A\} \cup \{<1,b> \mid b \in B \text{ and } b \ne \bot_B\} \cup \{\bot\}$. It is partially ordered by $x \le x'$ iff

1. $x = \bot$, or
2. $x = <0,x>$ and $x' = <0,x'>$ and $x \le_A x'$, or
3. $x = <1,x>$ and $x' = <1,x'>$ and $x \le_B x'$

A coalesced sum is a domain.

Proposition 19: Given domains A and B, let $[A \to B]$ be the set of all continuous functions from A to B with the partial order $f \leq g$ iff $(\forall a \in A)\ f(a) \leq_B g(a)$. $[A \rightarrow B]$ is called a *function space.* A function space is a domain with $(\forall a \in A)\ (\text{lub}(F))a = \text{lub}\{f(a) \mid f \in F\}$, and $\bot_{A \rightarrow B} = (\lambda a \in A.\ \bot_B)$.

Proposition 20: Let $f: A \otimes B \rightarrow C$. Then f is continuous iff f is continuous in its arguments separately, that is, iff $\lambda a.\ f(a, b')$ and $\lambda b.\ f(a', b)$ are continuous for all a′, b′.

Proposition 21: Define application ap: $[A \rightarrow B] \otimes A \rightarrow B$ by $ap(f, x) = f(x)$. Then ap is continuous with respect to the partial orders we have defined.

The theory of least fixed point semantics establishes the meaning of recursively defined functions. If the domains are modeled as sets, one can construct recursive definitions that do not uniquely define a function. If the domains are modeled as cpo's the theory:

1. guarantees that the recursive definition has at least one function satisfying it
2. provides a means for choosing a "best" function out of the set of all functions satisfying the recursive definition. The best function corresponds to an operational intuition about the definition where the definition is run as a program on a machine.

Theorem 22: *fixed point theorem for cpo's*

1. Every $f \in [A \rightarrow A]$ has a fixed point
2. There exists a continuous function Fix $\in [[A \rightarrow A] \rightarrow A]$ such that for all $f \in [A \rightarrow A]$, Fix(f) is the least fixed point of f.

This theorem means that

1. $f(\text{Fix}(f)) = \text{Fix}(f)$
2. $(\forall a \in A)\ f(a) = a$ implies $\text{Fix}(f) \leq a$

Proposition 23: Fix can be defined as $\text{Fix}(f) = \text{lub}(f^n(\bot))$, $0 \leq n \leq \infty$, where $f^0(\bot) = \bot$ and $f^{i+1}(x) = f(f^i(x))$, $i > 0$.

The definition of Fix $\in [A \rightarrow A] \rightarrow A$ is used to solve equations of the form $x = f(x)$ where $f \in [A \rightarrow A]$.

An induction principle is useful for reasoning about recursively specified functions. Since the meaning of a recursively defined function is the limit of the meanings of its finite subfunctions, if all the subfunctions have a property, then the least fixed point has it as well. The notion of "property" is formalized as an inclusive predicate, where a *predicate* is a (not necessarily continuous) function from any domain to the distinguished domain of boolean values, denoted bool. The domain bool is a flat domain with values TRUE, FALSE, and $\perp_{bool}$.

One example of a non-monotonic predicate is program-halts: D → bool. Let program-halts(x) equal TRUE if x≠⊥ and let program-halts(x) equal FALSE if x=⊥. Note that ⊥ ≤ n, but it is not the case that program-halts(⊥) ≤ program-halts(n). This particular predicate cannot be implemented on any computer; it is the halting problem of computability theory. This is one motivation of why all functions used in denotational semantics must be monotonic. Another example illustrates the non-monotonic predicate stong equality, ≡: D ⊗ D → bool. The predicate ≡ yields the value TRUE when both arguments are ⊥ and FALSE when exactly one argument is ⊥. In other words, x ≡ y iff x ≤ y & y ≤ x. Note that <⊥, d> ≤ <d, d>, but it is not the case that (⊥ ≡ d) ≤ (d ≡ d). This reflects the result that when x and y are computed as the result of a program, the relation ≡ is the notion of equivalence between programs, which is undecidable in general. The predicate, weak equality, =: D ⊗ D → bool yields the answer ⊥ whenever at least one of its arguments is ⊥. It is monotonic, in addition to continuous, if D is flat. A continuous predicate closely related to equality is δ: D → bool where δ(x) is TRUE if x ≠ ⊥ and δ(x) is ⊥ if x≡⊥. Define a computable equality eq as eq(x, y) equals TRUE if δ(x)≡ δ(y)≡TRUE and x≡y. It equals FALSE if δ(x)≡δ(y)≡TRUE and x≠y. And it equals ⊥ otherwise.

Definition 24: A predicate p: D → bool is an *inclusive predicate* iff for every chain C ⊆ D, if (∀c ∈ C) P(c) = TRUE, then p(lub(C)) = TRUE.

Definition 25: *The fixed point induction principle* is for a cpo D, a continuous function $F: D \rightarrow D$, and an inclusive predicate $p: D \rightarrow bool$, if:

1. $p(\perp)$ holds, and
2. for arbitrary $d \in D$, when $p(d)$ holds, then $p(F(d))$ holds

then $p(Fix(F))$ holds.

The really hard problem is determining whether a predicate is inclusive. This is especially important for the compiler correctness problem. A class of inclusive predicates is defined in [manna 72] as follows:

Proposition 26: A class <IP> of inclusive predicates can be defined as:

<IP>::= <IP> $\wedge$ <IP> | $(\forall x)$ <P>

<P>::= <P> $\vee$ <P> | $Q(x)$ | $F(f)(x) \leq G(f)(x)$ where x is a set domain variables, f is a recursively defined function, $Q(x)$ is a first order predicate, and $F(f)(x)$ and $G(f)(x)$ are function expressions using only f and x as free identifiers.

Thus, an inclusive predicate can be a universally quantified conjunction of disjunctions. It was shown above that the predicate $\equiv$ is not monotonic. We also have $x \equiv y$ iff $x \leq y$ & $y \leq x$, which is in the class of inclusive predicates. If $c \leq y$ where c is in chain C, then $lub(C) \leq y$. To show that the expression $F(f) \leq G(f)$ is inclusive, where $f = T(f)$ and F and G are continuous functionals (high order functions), we show $F(Fix(T)) \leq G(Fix(T))$ whenever $F(\perp) \leq G(\perp)$ and $(\forall f)$ $(F(f) \leq G(f) \supset F(T(f)) \leq G(T(f)))$. Assume F and G are strict so that the basis is satisfied. By induction, $(\forall i)$ $F(T^i(\perp)) \leq G(T^i(\perp))$. Furthermore, $G(T^i(\perp)) \leq G(Fix(T))$. Thus, $(\forall i)$ $F(T^i(\perp)) \leq G(Fix(T))$. This implies $lub(F(T^i(\perp))) \leq G(Fix(T))$. Because F is continuous, $F(lub(T^i(\perp)) \equiv F(Fix(T)) \leq G(Fix(T))$.

A contribution of this report will be to expand on the concept of inclusive predicates in order to define subdomains and quotients of domains.

Recursive definitions that specify functions were discussed above. Similarly, there are recursively defined domains (also called reflexive domains) of the form D = F(D). A solution to a recursively defined function f was achieved by treating the definition as an operational definition and recursively unfolding f's definition as needed. Similarly, the solution to a recursive domain definition is achieved by building a sequence of approximating domains. One particular solution method is called the *inverse limit construction*. We do not use the notions of a universal domain or a category-theoretic model in this paper.

The main result is that, for the recursive domain specification D = F(D), where F is an expression built with domain constructors, there is a cpo D_∞ that is isomorphic to $F(D_\infty)$. Furthermore, D_∞ is the least cpo. This is summarized in the following theorem.

> **Theorem 27:** There exists a unique minimal solution (up to isomorphism) to any system of equations defining domains recursively by expressions involving the operators $\otimes$, $\rightarrow$, $\oplus$, and $*$.

Some details of the inverse limit construction are presented below because these are used later to argue the proposed extension to interpretations.

> **Definition 28:** For cpo's A and B, a pair of continuous functions $\langle f: A \rightarrow B, g: B \rightarrow A \rangle$ is a *retraction pair* iff:
>
> 1. $g \circ f = id_A$
> 2. $f \circ g \leq id_B$
>
> f is called an *embedding* and g is called a *projection*. The function pair is also denoted $\langle f, g \rangle: A \leftrightarrow B$. The pair of continuous functions $\langle f, g \rangle$ is an *isomorphism pair* iff:
>
> 1. $g \circ f = id_A$
> 2. $f \circ g = id_B$

> **Proposition 29:** The composition $\langle f_2 \circ f_1, g_1 \circ g_2 \rangle$ of retraction pairs $\langle f_1: A \rightarrow B, g_1: B \rightarrow A \rangle$ and $\langle f_2: B \rightarrow C, g_2: C \rightarrow B \rangle$ is itself a retraction pair.

Proposition 30: An embedding (projection) has a unique corresponding projection (embedding).

Definition 31: The *reversal* for $\langle f, g\rangle: A \leftrightarrow B$ is $\langle g, f\rangle$ and is denoted $\langle f, g\rangle^R: B \leftrightarrow A$.

The reversal of a retraction pair might not be a retraction pair.

Proposition 32: For $f: A \leftrightarrow B$ and $g: B \leftrightarrow C$:

1. $(f \circ g)^R = g^R \circ f^R$
2. $(f^R)^R = f$

Definition 33: For retraction pairs $r = \langle f, g\rangle: C \leftrightarrow E$ and $s = \langle f', g'\rangle: C' \leftrightarrow E'$, let:

1. $r \otimes s$ denote: $\langle(\lambda(x, y). \langle f(x), f'(y)\rangle), (\lambda(x, y). \langle g(x), g'(y)\rangle)\rangle: C \otimes C' \leftrightarrow E \otimes E'$
2. $r \oplus s$ denote: $\langle(\lambda x.\ isl_{CC'}(x) \rightarrow inl_{EE'}\ (f(outl(x))),\ isr_{CC'}(x) \rightarrow inr_{EE'}(f'(outr(x)))),\ (\lambda x.\ isl_{EE'}(x) \rightarrow inl_{CC'}(g(outl(x))),\ isr_{EE'}(x) \rightarrow inr_{CC'}(g'(outr(x)))\rangle: C \oplus C' \leftrightarrow E \oplus E'$[6]
3. $r \rightarrow s$ denote: $\langle(\lambda x.\ f' \circ x \circ g), (\lambda y.\ g' \circ y \circ f)\rangle: (C \rightarrow C') \leftrightarrow (E \rightarrow E')$

For $D = F(D)$, the domain expression F determines both a construction for building a new domain $F(A)$ from an argument domain A and a construction for building a new retraction pair $F(r)$ from an argument retraction pair r. The retraction pair for flat domain D is (id_D, id_D), which is denoted $id_{D\leftrightarrow D}$.

Theorem 34: For any domain expression F and retraction pairs $r: A \leftrightarrow B$ and $s: B \leftrightarrow C$:

1. $F(id_{E\leftrightarrow E}) = id_{F(E)\leftrightarrow F(E)}$
2. $F(s) \circ F(r) = F(s \circ r)$
3. $(F(r))^R = F(r^R)$
4. if r is a retraction pair, then so is $F(r)$

[6]The notation $A \rightarrow B, C$ denotes a conditional expression, where A is a boolean expression. If A simplifies to TRUE then the conditional expression simplifies to B. If A simplifies to FALSE then the conditional expression simplifies to C. If A simplifies to $\perp$ then the conditional expression simplifies to $\perp$.

Definition 35: A *retraction sequence* is a pair $<\{D_n \mid n \geq 0\}, \{r_n: D_n \leftrightarrow D_{n+1} \mid n \geq 0\}>$ such that for all $n \geq 0$, D_n is a cpo and each r_n is a retraction pair. Denote each r_n pair as $<\iota_n : D_n \rightarrow D_{n+1}, j_n : D_{n+1} \rightarrow D_n>$.

Definition 36: Define $t_{mn} : D_m \leftrightarrow D_n$ as

1. $r_{n-1} \circ \ldots \circ r_m$, if $m < n$
2. $id_{D_m \leftrightarrow D_n}$, if $m = n$
3. $r_n^R \circ \ldots \circ r_{m-1}^R$, if $m > n$

Definition 37: Denote each t_{mn} as the pair $<\theta_{mn} : D_m \rightarrow D_n, \theta_{nm} : D_n \rightarrow D_m>$.

Proposition 38: For any retraction sequence and $m, n, k \geq 0$:

1. $t_{mn} \circ t_{km} \leq t_{kn}$
2. $t_{mn} \circ t_{km} = t_{kn}$, if $m \geq k$ or $m \geq n$
3. t_{mn} is a retraction pair when $m \leq n$

Definition 39: The *inverse limit* of a retraction sequence $<\{D_n \mid n \geq 0\}, \{<\iota_n, j_n> : D_n \leftrightarrow D_{n+1} \mid n \geq 0\}>$ is the set $D_\infty = \{(x_0, x_1, \ldots, x_n, \ldots) \mid$ for all $n > 0$, $x_n \in D_n$ and $x_n = j_n(x_{n+1})\}$. D_∞ is partially ordered by the relation : for all $x, y \in D_\infty$, $x \leq y$ iff for all $n \geq 0$ $prn(x) \leq_{D_n} prn(y)$ where $prn((x_0, x_1, \ldots)) = x_n$ (i.e., prn is the generalization of pr1 and pr2).

Theorem 40: D_∞ is a cpo.

Proposition 41: if domain expression F maps a cpo E to a cpo F(E) then the following pair is a retraction sequence: $<\{D_n \mid D_0 = \{\bot\}, D_{n+1} = F(D_n)$, for $n \geq 0\}, \{<\iota_n, j_n> : D_n \leftrightarrow D_{n+1} \mid \iota_0 = (\lambda x.\bot_{D_1}), j_0 = (\lambda x. \bot_{D_0}), <\iota_{n+1}, j_{n+1}> = F(<\iota_n, j_n>)$, for $n \geq 0\}>$

The inverse limit D_∞ exists for the retraction generated by F. Furthermore, D_∞ is isomorphic to $F(D_\infty)$ and D_∞ is the lub of the retraction sequence.

Definition 42: $<\phi, \psi> : D_\infty \leftrightarrow F(D_\infty)$ is defined as $lub_{m=0,\infty} (F(t_{m\infty}) \circ t_{\infty(m+1)})$

Theorem 43:

1. $<\phi, \psi>^R \circ <\phi, \psi> = id_{D_\infty \leftrightarrow D_\infty}$

2. $<\phi, \psi> \circ <\phi, \psi>^R = id_{F(D_\bot) \leftrightarrow F(D_\bot)}$

Chapter 6

Extending Interpretations to Include Domains and Domain Equations

6.1. Overview

In this chapter a many-sorted first-order theory is extended to allow products, sums, function spaces, and domain equations. The theory is augmented with a fixed set of operator symbols and axioms that specify these items. This is analogous to what Wand did in [wand 82a] where he extended a DLP theory to include tuples. It is assumed that both the source theory (implemented theory) and target theory (implementing theory) are developed from the theory schema that has incorporated products, sums, function spaces, and domain equations.

A structure is defined that assigns cpo's to sorts and assigns continuous functions to operator symbols.[7] In the following discussion the word "domain" will refer to the syntactic object constructed from sort symbols and domain constructors. The structure assigns a cpo, a semantic object, to each domain.

An interpretation is defined for the theory schema and then Theorem 4.1, defined in a previous chapter, is reconsidered for this particular theory schema and interpretation. The important questions that are considered include:

---

[7]The model of the theory of Scott domains forms a cartesian closed category with cpo's as objects and continuous maps as arrows. This is a cartesian closed category with function spaces (D→D) of the same cardinality as D.

1. what is the map from target states to source states given that the carriers are cpo's?
2. does the source structure, derived via the state map, preserve cpo's and continuity?
3. are there any changes to Wand's proof of Theorem 4.1?
4. are the fixed axioms that specify domains satisfied by the derived source structure?

6.2. Proof-Theoretic Version of Interpretations

In [wand 82a] and [enderton 72] a theory is a set of true formulas in some structure. In the extension of Wand's work presented here a theory will be a formal system consisting of (1) a language (set of symbols and a grammar), (2) axioms, and (3) rules of inference. If theory T_1 is interpreted in T_2, we require that both T_1 and T_2 be sound and closed under deduction.

The correctness conditions are satisfied by deductions in T_2. Because T_1 is closed under deduction, if the correctness conditions are satisfied all valid L_1-formulas are correctly implemented. See Appendix A for details.

6.3. Syntax and Structure of the Specification Language

In the following discussion a many-sorted first-order theory is extended to include domains. No claim is made that the choice of operators/axioms presented here is in some sense "best" or complete. The selection made is representative of axioms in current literature (e.g., [dybjer 83] [gordon 79a]). In practice, a set of "useful" and "efficient" axioms evolves when particular applications are considered and/or when software is developed to perform some theorem proving tasks.

6.3.1. Grammar and Fixed Symbols

Symbols

<sort symbols>
⊗
⊕
→
≃
λ
.
<constant symbols>
<n-place operator symbols>
<variable symbols>

Domains

<domain> ::= <sort symbol> | <domain> ⊗ <domain> |
<domain> ⊕ <domain> | <domain> → <domain>

Domain Equations

<domain equation> ::= <sort symbol> ≃ <domain>

Terms

<term: D_2> ::= <variable symbol: D_2> |
$\text{apply}_{D_1 \rightarrow D_2}$ (<term: $D_1 \rightarrow D_2$> , <term: D_1>) |
(<term: $D_1 \rightarrow D_2$>) (<term: D_1>)

<term: $D_1 \rightarrow D_2$> ::= <operator symbol: $D_1 \rightarrow D_2$> |
λ<variable symbol: D_1> . <term: D_2>

Atomic Formulas

<aform> ::= <term: bool>

Formulas

<form> ::= <aform> | ¬ <form> | <form> ⊃ <form>

The following are the fixed domains for the theory schema:
bool
nat
1

Let D, D_1, D_2, D_3, ... range over domains. Let c range over constant symbols, v range over variable symbols, and h range over n-place operator symbols. The domain over which the symbol ranges is called the signature of that symbol. If symbol v ranges over domain D we write v: D or $v \in D$. Each constant, variable, and operator symbol has a signature constructed from domains as follows:

$c: 1 \to D$
$v: D$
$h: D_1 \to D_2$

It is assumed below that when there are no parentheses to indicate the relative binding strength of the domain constructors, the constructor $\oplus$ has higher binding strength than $\otimes$, and $\otimes$ has higher binding strength than $\to$. The following are the fixed operator symbols for the theory schema:

$\neg$: bool $\to$ bool
$\supset$: bool $\otimes$ bool $\to$ bool
V: bool $\otimes$ bool $\to$ bool
&: bool $\otimes$ bool $\to$ bool
$cond_D$: bool $\otimes D \otimes D \to D$
$pair_{D_1D_2}: D_1 \to (D_2 \to (D_1 \otimes D_2))$
$pr1_{D_1D_2}: D_1 \otimes D_2 \to D_1$
$pr2_{D_1D_2}: D_1 \otimes D_2 \to D_2$
$outl_{D_1D_2}: D_1 \oplus D_2 \to D_1$
$outr_{D_1D_2}: D_1 \oplus D_2 \to D_2$
$inl_{D_1D_2}: D_1 \to D_1 \oplus D_2$
$inr_{D_1D_2}: D_2 \to D_1 \oplus D_2$
$isl_{D_1D_2}: D_1 \oplus D_2 \to$ bool
$isr_{D_1D_2}: D_1 \oplus D_2 \to$ bool
$id_D: D \to D$
$apply_{D_1 D_2}: (D_1 \to D_2) \otimes D_1 \to D_2$
$\circ_{D_1D_2D_3}: (D_1 \to D_2) \otimes (D_2 \to D_3) \to (D_1 \to D_3)$
$curry_{D_1D_2D_3}: (D_1 \otimes D_2 \to D_3) \to (D_1 \to D_2 \to D_3)$
$uncurry_{D_1 D_2 D_3}: (D_1 \to D_2 \to D_3) \to (D_1 \otimes D_2 \to D_3)$
$=_D: D \otimes D \to$ bool
TRUE: $1 \to$ bool
FALSE: $1 \to$ bool

Because operator symbols can be passed as parameters, and thus, treated as any other variable symbol, the apply operator is used to delimit terms. In the interpretation defined for predicate calculus [enderton 72], a term was identified in a string of symbols by scanning the string from right to left and finding the first function symbol. Obviously, this does not work for higher order operators. In the proposed theory schema here, application of an operator is explicitly specified with the apply operator or with parentheses.

In addition to delimiting terms, the apply operator can be used in defining the inverse operator for curry. From [dybjer 83] $\text{uncurry}_{ABC}(g) = \text{apply}_{BC} \circ \text{pair}_{(A\otimes B\to B\to C)(A\otimes B\to B)}(g \circ \text{pr1}_{AB}, \text{pr2}_{AB})$. Using this definition, uncurry(curry(f)) = f where f: $A\otimes B\to C$. Because $\text{apply}_{(A\otimes B\to C)(A\to B\to C)}(\text{curry}_{ABC}, \text{apply}_{BC} \circ \text{pair}_{(A\otimes B\to B\to C)(A\otimes B\to B)}(g \circ \text{pr1}_{AB}, \text{pr2}_{AB})) = g$, we also have curry(uncurry(g))=g.

6.3.2. Fixed Axioms

The following notation is used to simplify expressions. First, if the argument domain of an operator is a product (e.g., $A \otimes B$ is the argument domain of f where f: $(A \otimes B) \to C$), then the pair operator (or more commonly, angle brackets) may be omitted from a term involving that operator. That is, f(pair(a)(b)) may be written as f(<a, b>) or f(a, b). Furthermore, if all arguments are supplied to a curried operator the term may be written as if the operator is uncurried. Thus, pair(a)(b) may be rewritten as pair(a, b). Also, the notation ${}^{x}_{a}T$ denotes a term where all free occurrences of the variable x in the term T are replaced with a.

pair(pr1(x), pr2(x)) = x

pr1(pair(x, y)) = x

pr2(pair(x, y)) = y

outl(inl(x)) = x

$\text{outr}(\text{inr}(x)) = x$

$\text{isl}(\text{inl}(x)) = \text{TRUE}$

$\text{isr}(\text{inr}(x)) = \text{TRUE}$

$\text{isr}(x) \text{ iff } \neg\text{isl}(x)$

$\text{isl}(x) \supset \text{inl}(\text{outl}(x)) = x$

$\text{isr}(x) \supset \text{inr}(\text{outr}(x)) = x$

$\text{apply}_{A\,A}(\text{id}_A, x) = x$

$\text{apply}(\text{cond}, (\text{TRUE}, d_1, d_2)) = d_1$

$\text{apply}(\text{cond}, (\text{FALSE}, d_1, d_2)) = d_2$

$\text{apply}((\lambda x.T), a) = (\lambda x.\ T)(a) = {}^{x}_{a}T$

$\text{apply}((\lambda x.T), x) = T$ if x is not a free variable in T

$l{=}r\ \&\ F \supset {}^{l}_{r}F$

$f =_{A\to B} f'\ \&\ g =_{B\to C} g' \supset g \circ f =_{A\to C} f' \circ g'$

$f\colon A \to B\ \&\ g\colon B \to C\ \&\ h\colon C \to D \supset (h \circ g) \circ f =_{A\to D} h \circ (g \circ f)$

$f\colon A\to B \supset f \circ \text{id}_A = f\ \&\ \text{id}_B \circ f = f$

$f =_{A \otimes B\to C} g \supset \text{curry}(f) = \text{curry}(g)$

$A \approx A$

$A \approx B \supset B \approx A$

$A \approx B\ \&\ B \approx C \supset A \approx C$

$A \approx B \supset \exists\Theta_{AB}\colon\ A \to B$, where Θ_{AB} is bijective

$f\colon A\to B\ \&\ g\colon B \to A\ \&\ g \circ f = \text{id}_A\ \&\ f \circ g = \text{id}_B \supset A \approx B$

$(A \to B \to C) \approx (A \otimes B \to C)$

$\text{curry}(\text{uncurry}(g)) = g$

$\text{uncurry}(\text{curry}(f)) = f$

usual axioms for bool (boolean values) and nat (natural numbers)

6.3.3. Structure

The semantics for the language of the theory described above is given by a structure (function) named M. Prior to defining M some other definitions are in order.

Definition 1: A domain is called an *atomic domain* (also called a ground domain) if it is a sort symbol and does not appear on the left hand side of a domain equation.

Definition 2: A domain is called a *derived domain* if it is not an atomic domain.

If a derived domain is the left hand side of a domain equation, it is treated as an abbreviation for the domain on the right hand side.

$M(D) = \langle U_D^M, \leq_D^M \rangle$, where for the cpo assigned to domain D, U_D^M is the nonempty set and $\leq_D^M$ is the partial order on U_D^M. M(D) is flat if D is an atomic domain. Otherwise, the ordering is based on the domain constructors and is described in Chapter 5.

$M(v\colon D)\rho = \rho(v)$ where ρ: variables $\rightarrow U_D$

$M(h\colon D_1 \rightarrow D_2) = \rho(h) = h^M\colon U_{D_1} \rightarrow U_{D_2}$, such that h^M is continuous.

$M(bool) = \langle \{\perp_{bool}, TRUE, FALSE\}, \leq_{bool}^M \rangle$

$M(nat) = \langle \{\perp_{nat}, 1, 2, 3, \ldots\}, \leq_{nat}^M \rangle$

$M(1_D) = \langle \{\perp_D\}, \leq_D^M \rangle$

the logical symbols have the usual meanings

If D is an atomic domain then its structure is an ordinary set. The "lifting" construction which adds the bottom element to a set, is a common tool for converting a set into a cpo, in this case, a flat cpo.

6.4. The Interpretation

If both the source and target theories are constructed from the theory schema defined in the previous section, then there is an interpretation from the source theory to the target theory and correctness criteria for the interpretation such that the Implementation Theorem holds. In a later chapter, interpretation alternatives that may make the approach easier to use are discussed. Although the formal notation is rather tedious and the details can get messy, the concept of an implementation specification via interpretation between theories is straightforward and the work here tries to preserve the intent of [wand 82a]. Basically, a source object can be represented as any one of a subset of target objects and a particular source object can have many equally good target representations. It is a bit tricky with domains where, based on the composition of the source object, we restrict the type of target object representation. However, the partial order, bottom values, etc. associated with each domain need not concern the designer; they are used to give the theories a structure and are discussed in this paper in order to show that the Implementation Theorem holds with our extension to interpretation between theories. Define the interpretation I of L_1 in L_2 as follows:

1. for atomic domain s:

   a. assign to s a domain D that is constructed from atomic target domains, the symbol $\otimes$, and the symbol $\oplus$ (i.e., the domain constructor $\rightarrow$ is not allowed). Only domains specified as function spaces can be implemented as function spaces.[8] The interpretation is identity for the fixed domains bool, nat, and 1.

   b. create a formula named is-s with signature $D \rightarrow$ bool. This formula restricts the target domain to those elements that are legal representatives of domain s. For the distinguished fixed domains, bool, nat, and 1, is-s(d)=TRUE.

   c. assign to $=_s$ a formula with signature $D \otimes D \rightarrow$ bool. As in [wand 82a] this formula must specify an equivalence relation.

[8] There is an obvious exception to this where, under certain assumptions, an atomic domain can be interpreted as a function space. This is discussed in Chapter 8.

This formula specifies those target elements that are considered equivalent at the source level. For the distinguished fixed domains bool, nat, and 1, equality is interpreted as equality.

2. for derived domain D:

   a. if D = A ⊗ B and A and B are not derived from D then

      i. I(D) = I(A) ⊗ I(B)

      ii. is-D is defined as is-D(x) iff is-(A⊗B)(x) iff is-A(pr1(x)) & is-B(pr2(x)).

      iii. $I(=_D)$ is defined as $I(=_D)(x, y)$ iff $I(=_{A\otimes B})(x, y)$ iff $I(=_A)$(pr1(x), pr1(y)) & $I(=_B)$(pr2(x), pr2(y)).

   b. if D = A ⊕ B and A and B are not derived from D then

      i. I(D) = I(A) ⊕ I(B).

      ii. is-D is defined as is-D(x) iff is-(A⊕B)(x) iff (isl(x) ⊃ is-A(outl(x))) & (isr(x) ⊃ is-B(outr(x))).

      iii. $I(=_D)$ is defined as $I(=_D)(x, y)$ iff $I(=_{A\oplus B})(x, y)$ iff (isl(x) & isl(y) ⊃ $I(=_A)$(outl(x), outl(y))) & (isr(x) & isr(y) ⊃ $I(=_B)$(outr(x), outr(y))).

   c. if D = A → B and A and B are not derived from D then

      i. I(D) = I(A) → I(B)

      ii. is-D is defined as is-D(x) iff is-(A→B)(x) iff (is-A(a) ⊃ is-B(apply(x, a))) & ($I(=_A)$(a,a') ⊃ $I(=_B)$(apply(x, a), apply(x, a'))).[9]

      iii. $I(=_D)$ is defined as $I(=_D)(x, y)$ iff $I(=_{A\to B})(x, y)$ iff $I(=_A)$(a, a') ⊃ $I(=_B)$(apply(x, a), apply(y, a')).

   d. if D is recursively defined, say D = F(A, D) where F(A, D) is a term constructed from D, atomic domains which are represented by A, and domain constructors, then

      i. I(D) = D' where D' = F(I(A), D').

      ii. is-D is defined as is-D(x) iff is-F(A, D)(x). This formula is defined inductively from the domain construction F(A, D) using the definitions of is-(A⊗B), is-(A⊕B), and is-(A→B) above. Therefore, is-D is defined recursively. We prove that this predicate, defined by the recursive equation, exists.

      iii. $I(=_D)$ is defined as $I(=_D)(x, y)$ iff $I(=_{F(A, D)})(x, y)$. This formula is defined inductively from the domain construction F(A, D) using the definitions of $=_{A\otimes B}$, $=_{A\oplus B}$, and $=_{A\to B}$ above. Therefore, $I(=_D)$ is defined recursively and we prove that this predicate exists.

---

[9] It is assumed that all formulas are closed; the variables a and a' are universally quantified.

3. for each constant symbol c, $I(c: 1 \rightarrow D) = c^I: 1 \rightarrow A$ where $A = I(D)$.
4. for each variable symbol v, $I(v: D) = v^I: A$ where $A = I(D)$.
5. for each n-place operator symbol h, $I(h: D_1 \rightarrow D_2) = h^I: A \rightarrow B$ where h^I is a term in L_2, $A = I(D_1)$, and $B = I(D_2)$.
6. I is identity on the logical operators and constants.
7. to each fixed polymorphic operator p, p ∈ {pair, pr1, pr2, cond, outl, outr, inl, inr, isl, isr, id, °, curry, uncurry, apply}, assign the same operator in the target theory with the signature of a domain isomorphic to the interpreted source signature.

Above, the interpretation of domains is described as a "bottom-up" process; the interpretation of a non-recursive domain is the interpretation of its isomorphic construction of atomic domains. We shall prove in the following sections that because of our formulations of is-D and $I(=_D)$, the isomorphisms specified at the source level are preserved in the implementation. However, it may be more natural for a designer to specify a domain interpretation irrespective of its underlying composition; i.e., it may be more natural to use a "top-down" approach. Then it would fall upon the designer to show that the domain interpretation preserves the domain's underlying composition as specified by the source domain equations. We will not discuss this here, but rather, discuss this alternative approach to interpretation in a later section. For now, assume the domain interpretations are constructed in a bottom-up process.

Terms and formulas are basically interpreted by interpreting each symbol in the expressions. The preamble is added as in [wand 82a] and serves the purpose of restricting target elements to those elements that are legal representatives of source domains.

However, the preamble defined below differs from the preamble in [wand 82a]. In [wand 82a] the preamble of a formula is defined in terms of the set of free variables in the formula. In the approach proposed here, the constant, variable, and operator symbols have equal status. A set of "free symbols", FS, is defined inductively from the inductive formula definition as follows:

1. FS(x) = {x} where x is a constant, variable or operator symbol
2. $FS(apply\ (t_1, t_2)) = \{apply\} \cup FS(t_1) \cup FS(t_2)$
3. $FS((t_1)(t_2)) = FS(t_1) \cup FS(t_2)$
4. $FS(\lambda v.\ t) = FS(t) - \{v\}$
5. $FS(\neg f) = FS(f)$
6. $FS(f_1 \supset f_2) = FS(f_1) \cup FS(f_2)$

The preamble of formula α is a formula pre(α) and is defined by:

$$pre(\alpha) = is\text{-}D_1(I(x_1))\ \&\ \ldots\ \&\ is\text{-}D_n(I(x_n))$$

where $FS(\alpha) = \{x_1, \ldots, x_n\}$ & for $1 \le i \le n$, x_i: D_i

The interpretation is defined for formula α as $(pre(\alpha) \supset I(\alpha))$ where I is defined on terms and formulas as:

1. $I(apply(t_1, t_2)) = I(apply)(I(t_1), I(t_2))$
2. $I((t_1)(t_2)) = (I(t_1))(I(t_2))$
3. $I(\lambda v.\ t) = \lambda I(v).\ (is\text{-}D(v) \supset I(t))$, where v: D
4. $I(\neg f) = \neg I(f)$
5. $I(f_1 \supset f_2) = I(f_1) \supset I(f_2)$

Notice that items 4 and 5 result from the fact that $I(\neg) = \neg$, $I(\supset) = \supset$, and I(bool)= bool.

For example, the interpretation of term $apply_{A \to B}(f, x)$ is (pre $\supset$ I(apply)(I(f), I(x))) where the preamble pre is defined as is-(A→B)(I(f))& is-A(I(x)) & is-((A→B)⊗A→B)(I($apply_{A\ B}$)). The preamble can be simplified to is-A(I(x)) after correctness conditions, discussed in the next section, are satisfied.

At first glance, the typing of interpreted symbols appears overly complicated. Why not simply let the signature of a source symbol interpretation be the interpretation of the source symbol's signature? In practice, it may be desirable to interpret an object in a source domain by referring to a finer composition of a target domain than is indicated by the domain interpretation. In effect, this is not really different than a direct interpretation of a source signature because domain equations can be treated as domain abbreviation definitions. The retraction pairs that are used to coerce an object in one domain to an object in an isomorphic domain implicitly exist in expressions. While the retraction pairs exist at the structure level, they do not appear in the theories. If an object a is in domain A and A is isomorphic to domain B, then at the theory level a has both types A and B. At the structure level there exists retraction pairs (which are isomorphisms) between the cpo's for A and B such that an object in the cpo for A can be coerced into an object in the cpo for B, and vice versa.

Further, a curried application operation in the source may be implemented by an uncurried term. Abbreviate $apply_{B\ C}$ ($apply_{A\ (B \to C)}$ (f, a), b) as $apply_{A\ B\ C}$ (f, a, b). Now $apply_{A\ B\ C}$ (f, a, b) = $apply_{A \otimes B\ C}$(uncurry(f), pair(a, b)). This is useful if I(A) → I(B) → I(C) does not exist in the target theory, but I(A) ⊗ I(B) → I(C) does exist. Similarly, one can curry or uncurry a lambda term.

6.5. The Correctness Criteria

The following correctness conditions are proposed for the interpretation I:

1. $T_{target} \vdash (\exists x)(\text{is-D}(x))$ for each atomic source domain D.
2. $T_{target} \vdash \text{is-A}(x) \supset \text{is-B}((I(f))(x))$ for each source operator symbol f with signature A→B.[10]
3. $T_{target} \vdash I(x =_D x)$ for each atomic source domain D.
4. $T_{target} \vdash I(x =_A y \supset (f)(x) =_B (f)(y))$ for each source operator symbol f with signature A→B.
5. $T_{target.} \vdash \text{pre}(F) \supset I(F)$ for each source axiom F.

If f is an operator symbol in some source axiom, then conditions 2 and 4 for I(f) will be stated in the preamble of the interpreted axiom; the conditions are stated explicitly in the list of assumptions about the formula interpretation. These conditions require that interpreted operators take source representative arguments into source representative results and that interpreted operators take equivalent arguments into equivalent results. The interpreted axioms, which incorporate the preambles, must be deducible in the target theory. This is in contrast to [wand 82a] where the assumptions about operator symbols are not listed in the preamble. In this report an operator symbol can be passed as an argument to another operator, and thus, is treated as any other symbol in a formula.

Furthermore, the correctness conditions above differ from [wand 82a] in that conditions 1 and 3 are stated in terms of atomic domains, rather than sort symbols. We show later by induction that those conditions hold for any domain.

[10]Actually, the expression is-B((I(B))(x)) should be written $\text{is-B}(\theta_{D(I[B])}(\text{apply}_{CD}(I(f), \theta_{I(A)C}(x))))$ where I(f): C→D such that C= I(A) and D= I(B). However, domain equations in some sense denote domain abbreviations and it is assumed that the appropriate domain coercions take place.

6.6. Discussion About Interpretation

In an attempt to define an implementation as an interpretation, we have limited (perhaps severely limited) the kinds of relationship that can hold between source and target domains. In our proposal, the source and target domains must be very similar in structure. More general relationships have been described to show (1) that a denotational and operational semantics of a language are equivalent [stoy 77], and (2) that a direct semantic definition may be implemented as a continuation semantic definition [reynolds 74]. Certain predicates, called inclusive, ω-inductive, or directed complete predicates, describe the general relationships. However, in general, it is difficult to specify the predicates and show they exist, even for small, scale-downed problems. Furthermore, such methods do not address the general problem of changing the representation of a programming language (i.e., translating the source programming language into a target programming language) for large languages.

We suggest that for clarity of design and practicality of proof, the implementation, which defines a change of representation, (1) be specified as an interpretation (mapping) and (2) proceed in a sequence of steps, each step specifying a small change in representation. It is proposed that the predicates defined in the interpretation be restricted so that the designer does not have to show that the predicates exist. Presently, in the compiler design problem it appears natural to, for example, represent a source environment by some target environment and a source continuation by some target continuation. This is not to minimize the significance of more general relationships. We are not ignoring the possibility that the source/target relationships proposed here might be too restrictive for some applications. For the compiler design problem, it does not seem unreasonable to specify both languages with denotational semantics (e.g., as in [polak 80]), and it does not seem unreasonable to specify both source and target programming languages with continuation semantics if one of the languages is specified with continuation semantics. A considerable amount of progress has been achieved if the compiler design problem can be dealt with at this level because:

1. the informal process of changing representation via mental translation and comparison closely corresponds to the formal process of defining an interpretation.
2. the implementation of the source programming language syntax is treated in the same way as the implementation of the source programming language semantics.
3. the correctness proof is based on a structural induction argument.
4. the induction steps required to show that any source program is correctly implemented are implicitly handled by the interpretation, and thus, can be mechanized as a translation.
5. the correctness proof is systematically broken down into small subproofs. In particular, a change in the source programming language specification will not require a completely new correctness proof, but rather, only those subproofs effected by the specification change will have to be redone.
6. the proofs are done syntactically, as deductions, in the target theory.

Ideally, there should be no restriction on the source and target; in some circumstances it may be desirable to specify them independently. However, today's technology does not provide a practical way of organizing and carrying out such correctness proofs for large problems, independent of the style of semantics used. In this research, we step back and examine the problem with the goal of mirroring the design process in a formal way and carrying out the verification in a practical way. We admit that we cannot adequately deal with those situations with very dissimilar source and target domains in the specifications. It is a goal that the restrictions introduced here are ones that designers can live with.

6.7. Derivation of the Source Structure

Theorem 4.1 gives the 'model construction' result for interpretations. It states that the source structure can be derived from the target structure such that the interpretation of any true source formula is true and any source formula whose interpretation is true is also true. In other words, the source object behavior can be perceived by looking at the behavior of its implementation

The interpretation gives a syntactic translation of the source theory language into target theory language. The interpretation, specification language syntax, axioms and inference rules give a dangerous illusion of precision. Structures are used to give the syntactic system an unambiguous meaning. Once it is shown that the syntactic system behaves in the intended manner via the structure, we can operate totally within the syntactic system. Even though correctness proofs are done within the syntactic system, at any time the meaning of a formula can be derived by applying the structure. It is shown here that the syntactic system we have defined behaves in the intended manner (i.e., satisfies the Implementation Theorem) via the structure.

The map J from target states to source states is defined as in [wand 82a] with the exception that we do not use the 'undefined value' e_{σ}. Particular values in the target state that are not legal representations of source values are eliminated from domains under consideration. In particular, let T be the target structure and S the source structure. If $U^T_{I(D)}$ is the carrier for the interpretation of source domain D in the target structure, let $U^T_{is\text{-}D}$ be that subset of $U^T_{I(D)}$ where all values satisfy the formula is-D. Further, let $\simeq_D$ stand for the interpretation of $=_D$, where the subscript may be omitted if the context is clear. Then $U^T_{\simeq} = \{<x, y> \mid x, y \in U^T_{is\text{-}D} \; \& \; x \simeq^T_D y\}$, where T and D are omitted if the context is clear; $U_{\simeq}$ is an equivalence relation on $U_{is\text{-}D}$. The expression

$U_{is\text{-}D}/U_{\equiv}$ is used to denote the quotient of $U_{is\text{-}D}$. The source carriers are derived from the target by letting the source carrier for domain D be the quotient, defined by $\equiv$, of the is-D subset of the target carrier for I(D). That is, $U^S_D = U_{is\text{-}D}/U_{\equiv}$ where $U_{is\text{-}D}/U_{\equiv} = \{U_x \mid U_x \subseteq U_{is\text{-}D}$ & for some $d \in U_{is\text{-}D}$, $U_x = \{d' \mid \langle d, d'\rangle \in U_{\equiv}\}\}$. We will show later that $U_{is\text{-}D}/U_{\equiv}$ is a "good" carrier; the carrier together with the partial order specified defines a cpo.

Define J_D: $U^T_{is\text{-}D} \rightarrow U^S_D$ as $J_D(v) = [v]_{\equiv}$ and $J_D(\perp^T_{is\text{-}D}) = \{\perp^S_D\}$. Define map J from target states to source states as $J\rho x = J_{\sigma}(M(I(x))(\rho))$ where symbol x has signature σ. Source operations are derived as before, h^S: $U^S_A \rightarrow U^S_B$: $[a] \rightarrow [I(h)^T(a)]$ because we have is-(A→B)(I(h)) and this implies $a \equiv a' \supset (I(h))(a) \equiv (I(h))(a')$. In the discussion below, issues are addressed that concern the use of cpo's in the structure rather than sets.

Define the partial order for cpo's assigned to the source domains as follows:

1. $\leq_{is\text{-}D}$ (x, y) iff is-D(x) & is-D(y) & $\leq_{I(D)}$(x, y)
2. $\leq_{\equiv}$ is defined by the following where x and y are in $U_{is\text{-}D}$, [x] and [y] are in $U_{is\text{-}D}/U_{\equiv}$, and $a \in [x]$ means a is a member of the equivalence class [x]:
   $[x] \leq [y]$ iff $(\forall a \in [x])\ (\exists\ b \in [y])\ a \leq b\ \&\ (\forall b \in [y])(\exists\ a \in [x])\ a \leq b$

The first definition follows immediately from the fact that the is-D domain is a subset of the I(D) domain. The second definition is a weaker partial order than one usually defined for quotients in which two equivalence classes are ordered if and only if every class element is ordered with *every* element of the other class (i.e. $[x] \leq [y]$ iff $x \leq y$). The weaker partial order defined above models our notion of implementation. The definition states that two classes are ordered if and only if every element in a class is ordered with at least one element in the other class; if a source representative value has a better approximation then an equivalent value has a better approximation. A property follows from the interpretation and this definition. The property states that all elements in an equivalence class are unordered; they depict the same "level of approximation." Any member of an equivalence class is a good representative for the associated source object; any representative approximation is ordered in "lockstep" with any other equally good representative approximation.

While the properties correspond to the intuitive concept of implementation, we still have to show that such an order exists for any domain construction and that this is a partial order. We will also show that with this partial order definition we can define the lub of any chain in a quotient domain. This is discussed in the following section entitled, "Modelling the Quotient as a cpo."

6.7.1. Is the Quotient of a Source Representative Subset a Domain?

The carrier for source domain D is constructed by taking a quotient of the source representative subset of the carrier for the interpretation of D. The quotients and subsets are defined by predicates $=_D$ and is-D, respectively. For atomic source domains, the predicates are specified by the designer/implementer. For derived source domains, the predicates are defined in the prescribed manner above and listed below for convenience:

1. is-$(A \otimes B)(x)$ iff is-$A(pr1(x))$ & is-$B(pr2(x))$
2. is-$(A \oplus B)(x)$ iff $(isl(x) \supset$ is-$A(outl(x)))$ & $(isr(x) \supset$ is-$B(outr(x)))$
3. is-$(A \to B)(x)$ iff (is-$A(a) \supset$ is-$B(x(a)))$ & $(a =_A a' \supset x(a) =_B x(a'))$
4. $x =_{A \otimes B} y$ iff $pr1(x) =_A pr1(y)$ & $pr2(x) =_B pr2(y)$
5. $x =_{A \oplus B} y$ iff $(isl(x)$ & $isl(y) \supset outl(x) =_A outl(y))$ & $(isr(x)$ & $isr(y) \supset outr(x) =_B outr(y))$
6. $x =_{A \to B} y$ iff $a =_A a' \supset x(a) =_B x(a')$

Item 1 means that a target tuple is a source representative if and only if each projection of the tuple is a source representative. Similarly, in item 4, two tuples are equivalent if and only if their respective projections are equivalent. In items 2 and 5, a value in a sum domain is identified as belonging to either the left or right domain, and then the predicate associated with the identified domain is applied. Items 6 and 3 concern function spaces. Item 6 states that two functions are equivalent if and only if they take equivalent arguments into equivalent results. Item 3 states that a target function is a source representative if and only if (1) it takes source representative arguments into source representative results and (2) it takes equivalent arguments into equivalent results.

There is also the potential for recursive predicates because if source domain D is isomorphic to F(D), then by definition is-D iff is-F(D), and is-F(D) is defined in terms of is-D. Also, we have $=_D$ iff $=_{F(D)}$ and $=_{F(D)}$ is defined in terms of $=_D$.

The main problem is proving the existence of the recursive predicates -- a nontrivial problem. First consider nonrecursive predicates. The predicates must be inclusive if they are used to define cpo's, where predicate p is *inclusive* if for chain A, p(A) implies p(lub(A)). Informally, if the predicates are inclusive then the source representative target domains contain all the values we are interested in; they contain the limit of any source representative approximation.

If D is an atomic source domain then I(D) is an expression constructed from atomic target domains, the product constructor, and the sum constructor. In this case, I(D) is a flat cpo. Any subset or quotient on I(D) will also result in a flat cpo; the subsets and quotients will contain the limits of any subset or quotient chain. In other words, atomic domains and products and sums of atomic domains are treated as sets and we can define arbitrary predicates on sets.

From this discussion it is apparent why the interpretation of atomic domains was restricted. It is not clear how to define inclusive predicates on function spaces. At an intuitive level, if the source domain is atomic (the specification does not indicate how the domain is constructed) then it makes sense that the carrier derived from the interpretation be a flat cpo. It also might make sense that its carrier be "non-flat" if we could safely define function space predicates; that is, impose a complex structure on the source domain that is not indicated in the source theory.

At a practical level, if the designer decides that a source domain should be interpreted as a function space then the source domain can be "refined" by specifying it isomorphic to some source function space. It is not clear at this time whether this would prohibit the applicability of this approach. It is difficult to think of a compiler design problem where it would be impossible to define an interpretation if the semantics for the source and target languages are written in the same style. This should be investigated in the future.

If D is a non-recursive derived domain constructed from domains A and B, then is-D and $=_D$ are inclusive assuming the predicates for A and B are inclusive. Consider D = A⊗B. Let C={<a_1, b_1>, <a_2,b_2>, ...} be a chain in D. Then C_1={a_1, a_2, ...} is a chain in A and C_2={b_1, b_2, ...} is a chain in B. Assume is-A and is-B are inclusive. Then we have is-A(C_1) ⊃ is-A(lub(C_1)) and is-B(C_2) ⊃ is-B(lub((C_2)). We also have lub(C)=<lub(C_1), lub(C_2)>. This and the definition of is-D gives us is-D(C) ⊃ is-D(lub(C)); i.e., is-D is inclusive. Similarly, if $=_A$ and $=_B$ are inclusive then $=_D$ is inclusive.

The argument is similar for the sum and function space construction. Briefly, consider the function space construction, D=A→B. Let $F=\{f_1, f_2, \ldots\}$ be a chain in D. Let $F(a)=\{f(a) \mid f \in F\}$. We know $(\forall a \in A)(lub(F))(a) = lub(F(a))$. By assumption, is-B and $=_B$ are inclusive. We have is-B(F(a)) ⊃ is-B(lub(F(a))) and lub(F(a))=(lub(F))(a). Similarly, $F(a) =_B F(a') \supset (lub(F))(a) =_B (lub(F))(a')$. This, together with the definition of is-D, gives us that is-D is inclusive.

From the discussion above, we determined that the predicates defined on the non-recursive source domain interpretations are inclusive where a predicate for a derived source domain is defined in terms of predicates on its constituent domains. But, how is the existence of recursive predicates justified where recursive predicates result from the interpretations of recursive source domains? From Scott's work, we can deal with recursive functions and domains. That work depends on properties of monotonicity and continuity, and generally, both do not apply to predicates.

It is very difficult to come up with a nontrivial equation over predicates which does not have a solution. The first illustration of such counterexamples can be found in [mulmuley 85]. The counterexamples involve a subtle use of self-application. So, even though these predicates are not generally found, they do exist, and an argument must be made justifying the existence of any predicate. A key point is that the predicate existence problem is very sensitive to the domain construction and "there cannot be a rich enough purely syntactic language such that any predicate expressed in that language exists [mulmuley 85]."

The approach proposed in this paper is to allow a small set of predicates derived from the domain construction that are monotonic and inclusive. Thus, the designer does not have to prove the existence of the predicates defined in the interpretation, but the designer is restricted in the ways the interpretation can be specified. This paper proposes a particular set of restrictions -- others may be defined in the future.

6.7.1.1. The Existence of Predicates in the Interpretation

It must be shown that the definitions (possibly recursive) for $=_D$ and is-D exist. It is assumed from the discussion above that the predicates are inclusive for flat or non-recursive $U_{I(D)}$. [reynolds 74] and [milne 76] discuss techniques to prove predicate existence. These techniques are generalized and made systematic in [mulmuley 85]. The techniques in [reynolds 74] for proving recursive relations are modified here for proving recursive predicates. This modification is essentially a simplification of the technique in [mulmuley 85].

The basic idea is this: given the least domain D satisfying D = T(D), where T is a domain constructor, and a predicate P on domain D such that P = w(P), we want to show that P exists. For the interpretation proposed in this paper, the predicate P will be either $=_D$ or is-D and w is restricted to a predicate transformation based on the domain D.

The construction of the solution for P = w(P) is based on the inverse limit of the retraction sequence $<\{D_n \mid D_0 = \{\perp\}, D_{n+1} = T(D_n) \text{ for } n \geq 0\}, <i_n, j_n>: D_n \leftrightarrow D_{n+1} \mid i_0 = (\lambda x. \perp_{D_1}), j_0 = (\lambda x. \perp_{D_0}), <i_{n+1}, j_{n+1}> = T(<i_n, j_n>) \text{ for } n \geq 0\}>$. The inverse limit is the least fixed point satisfying T. The retraction sequence forms the following chain: $\perp \leq T(\perp) \leq T^2(\perp) \leq \ldots$. The technique in this section is to build the following chain of domain-predicate pairs beginning with $<\perp, \{\perp\}>$, where $\perp$ denotes the domain of one value, namely the bottom element, and $\{\perp\}$ is the trivial predicate on domain $\perp$: $<\perp, \{\perp\}> \leq <T(\perp), w\{\perp\}> \leq <T_2(\perp), w^2\{\perp\}> \leq \ldots$. First of all, $<T^n(\perp), w^n\{\perp\}> \leq$

$<T^{n+1}(\bot), w^{n+1}\{\bot\}>$ means that there is a retraction pair $<i_n, j_n>$: $T^n(\bot) \leftrightarrow T^{n+1}(\bot)$ such that $i_n(w^n\{\bot\}) \subseteq w^{n+1}\{\bot\}$, and $j_n(w^{n+1}\{\bot\}) = w^n\{\bot\}$. The limit of the chain should be $<D_\infty, P_\infty>$ where D_∞ is the least solution of $D = T(D)$, and P_∞ is a predicate on D_∞ such that it satisfies the equation $P = w(P)$. Both $T^n(\bot)$ and $w^n\{\bot\}$ must be closed under the lub operation. That is, the predicate $w^n\{\bot\}$ must be inclusive.

This is just a sketch of the method and as Mulmuley says in [mulmuley 85], the proofs become quite complicated when the details are filled in. Mulmuley proposes existence proofs that are mechanizable. However, these are complex and once the existence proofs are done, the correctness problem described in this research still remains. The more complex the predicates are, the harder it is to do the correctness proof. This paper proposes an alternative to doing both difficult existence proofs and a hard correctness proof for each implementation verification by investigating (relatively) simple predicate transformations that result in correctness proofs based on implicit structural induction. We basically follow Reynold's scheme where the existence proofs are simpler, but the methods are more restricted in their applicability. The methods are more restricted, but the resulting correctness proofs are understandable and manageable. If the designer follows the interpretation procedure of this paper, he/she can assume that all the predicates in the interpretation are good from the results in this section.

First, we show that there exists a P_∞ for the problem described above. The predicate P_∞ is inclusive and is the solution to $P = w(P)$. Then, we show that the predicate transformations defined in the interpretation satisfy the properties necessary to ensure a solution.

First of all, $w^n\{\perp\}$ is inclusive by the following induction argument. Let X be a chain in $w^n\{\perp\}$. We claim that for all n, lub(X) $\in w^n\{\perp\}$. The basis is trivially true because $w^0\{\perp\} = \{\perp\}$. The induction hypothesis is: X is a chain in $w^n\{\perp\}$ implies lub(X) is in $w^n\{\perp\}$. The proof proceeds as follows: if X is a chain in $w^{n+1}\{\perp\}$ then by the ordering, $j_n(X)$ is a chain in $w^n\{\perp\}$. By the induction hypothesis, lub(j_n(x)) is in $w^n\{\perp\}$. Because j_n is continuous, j_n(lub(X)) is in $w^n\{\perp\}$. Applying i_n, $i_n \circ j_n$(lub(x)) is in $i_n(w^n\{\perp\})$. By the retraction properties, lub(X) is in $w^{n+1}\{\perp\}$ and we are finished; for all n, $w^n\{\perp\}$ is inclusive.

We claim that $P_\infty = \{(x_0, x_1, \ldots) \mid \text{for all } n > 0, x_n \in w^n(\perp) \text{ and } x_n = j_n(x_{n+1})\}$. P_∞ is inclusive if for any chain $C = \{c_0, c_1, \ldots\}$ in P_∞ the lub(C) is in P_∞. Let $c_i = (x_{i0}, x_{i1}, \ldots)$. Then $c_i \leq c_{i+1}$ iff for all $n \geq 0$, $x_{in} \leq x_{(i+1)n}$. Let $C_n = \{x_{in} \mid i \geq 0\}$. Then C_n is a chain in $w^n(\perp)$ with lub(C_n) also in $w^n(\perp)$. We also have $j_n(\text{lub}(C_{n+1})) = \text{lub}(j_n(C_{n+1})) = \text{lub}(j_n(x_{i(n+1)}) \mid i \geq 0\} = \text{lub}\{x_{in} \mid i \geq 0\} = \text{lub}(C_n)$. Therefore, (lub($C_0$), lub($C_1$), ...) belongs to P_∞ and it is lub(C).

We claim $P_\infty = w(P_\infty)$. Let $<i_{n\infty}, j_{n\infty}>: T^n(\perp) \leftrightarrow D_\infty$. $<D_\infty, P_\infty>$ is in the domain-predicate pair chain because

1. $j_{n\infty} \circ i_{n\infty} (T^n(\perp)) = T^n(\perp)$
2. $i_{n\infty} \circ j_{n\infty} (D_\infty) = D_\infty$
3. $j_{n\infty} (P_\infty) = \{j_n \circ \ldots \circ j_\infty (x_0, x_1, \ldots) \mid x_n \in w^n(\perp) \,\&\, x_n = j_n(x_{n+1})\}$
   $= \{x_n \mid x_n \in w^n(\perp)\}$
   $= w^n(\perp)$
4. $i_{n\infty} (w^n(\perp)) = \{ i_\infty \circ \ldots \circ i_n (x_n) \mid x_n \in w^n (\perp)\}$
   $= \{ x \mid x \in D \subseteq w^\infty(\perp)\}$
   $\subseteq w^\infty(\perp) = P_\infty$

Furthermore, there is a retraction pair that is also an isomorphism pair $<\Phi, \Psi>$:

$D_\infty \leftrightarrow T(D)_\infty$. We know $\Psi(w(P_\infty)) = P_\infty$ and $\Phi(P_\infty) \subseteq w(P_\infty)$. We conclude $w(P_\infty) = \Phi\,(P_\infty)$ from $\Phi \circ (\Psi \circ w(P_\infty)) = \Phi(P_\infty) = (\Phi \circ \Psi) \circ w(P_\infty) = w(P_\infty)$. Therefore, $<\Phi, \Psi>: P_\infty \leftrightarrow w(P_\infty)$ and P_∞ is a solution. Finally, P_∞ is the least solution for the same reason D_∞ is; they are created with the same sequence of retractions.

So, now we must show that for all the predicate transformers w defined in the interpretation and for domain constructors T, $<D_1, P_1> \le <D_2, P_2>$ implies $<T(D_1), w(P_1)> \le <T(D_2), w(P_2)>$. If this property holds for all the predicate transformers, then the domain-predicate pair chain can be constructed as in the argument above and this chain has a limit based on the inverse limit construction.

Define the domain constructors and their corresponding predicate transformers as follows:

1. $T_\otimes(D) = T_1(D) \otimes T_2(D)$

   $w_\otimes(P) = \{<x, y> \mid x \ne \bot \;\&\; y \ne \bot \;\&\; x \in w_1(P) \;\&\; y \in w_2(P)\} \cup \{\bot\}$

   $w_{2\otimes}(P) = \{<<x, y>, <x', y'>> \mid x \ne \bot \;\&\; y \ne \bot \;\&\; x' \ne \bot \;\&$
   $y' \ne \bot \;\&\; <x, x'> \in w_1(P) \;\&\; <y, y'> \in w_2(P)\} \cup \{<\bot, \bot>\}$

2. $T_\oplus(D) = T_1(D) \oplus T_2(D)$

   $w_\oplus(P) = \{<0, x> \mid x \ne \bot \;\&\; x \in w_1(P)\} \cup \{<1, x> \mid x \ne \bot \;\&\; x \in w_2(P)\} \cup \{\bot\}$

   $w_{2\oplus}(P) = \{<0, <x, x'>> \mid <x, x'> \ne \bot \;\&\; <x, x'> \in w_1(P)\} \cup$
   $\{<1, <x, x'>> \mid <x, x'> \ne \bot \;\&\; <x, x'> \in w_2(P)\} \cup \{\bot\}$

3. $T_\rightarrow(D) = T_1\,(D) \rightarrow T_2\,(D)$

   $w_\rightarrow(P) = \{f \mid x \in w_1(P) \supset f(x) \in w_2(P)\}$

   $w_{1\rightarrow}(P) = \{f \mid <x, x'> \in w_1(P) \supset$
   $<f(x), f(x')> \in w_2(P)\}$

   $w_{2\rightarrow}(P) = \{<f, g> \mid <x, x'> \in w_1(P) \supset <f(x), g(x')> \in w_2(P)\}$

4. $T_{id}(D) = D$

   $w_{id}(P) = P$

Assuming that the predicate transformers satisfy the property above, the structures for is-D and $=_D$, constructed from any domain D, are given as follows:

1. M(is-$T_\otimes$(D)) is defined as $w_\otimes$(M(is-D))
2. M($=_{T_\otimes(D)}$) is defined as $w_{2\otimes}$(M($=_D$))
3. M(is-$T_\oplus$(D)) is defined as $w_\oplus$(M(is-D))
4. M($=_{T_\oplus(D)}$) is defined as $w_{2\oplus}$(M($=_D$))
5. M(is-$T_\rightarrow$(D)) is defined as $w_\rightarrow$(M(is-D)) $\cap$ $w_{1\rightarrow}$(M($=_D$))
6. M($=_{T\rightarrow(D)}$) is defined as $w_{2\rightarrow}$(M($=_D$))

Now, each of the predicate transformers is examined. First consider $w_\otimes$. We must show that $<D_1, P_1> \leq <D_2, P_2>$ implies $<T_\otimes(D_1), w_\otimes(P_1)> \leq <T_\otimes(D_2), w_\otimes(P_2)>$ with the assumption that for n = 1, 2, $<D_1, P_1> \leq <D_2, P_2>$ implies $<T_n(D_1), w_n(P_1)> \leq <T_n(D_2), w_n(P_2)>$. It is already known that $T_\otimes(D_1) \leq T_\otimes(D_2)$. So, the discussion focuses on predicate part of the domain-predicate pair. The two-part proof proceeds as follows, where $<i_\otimes, j_\otimes>: T_\otimes(D_1) \leftrightarrow T_\otimes(D_2)$, $<i_1, j_1>: T_1(D_1) \leftrightarrow T_1(D_2)$, $<i_2, j_2>: T_2(D_1) \leftrightarrow T_2(D_2)$, $i_\otimes = \lambda(x, y). <i_1(x), i_2(y)>$, and $j_\otimes = \lambda(x,y). <j_1(x), j_2(y)>$:

1. $j_\otimes(w_\otimes(P_2))$
   $=\{j_\otimes <x, y> \mid x \neq \perp, y \neq \perp, x \in w_1(P_2), y \in w_2(P_2)\} \cup \{j_\otimes(\perp)\}$
   $=\{<j_1(x), j_2(y)> \mid \ldots\} \cup \{\perp\}$
   $=\{<x, y> \mid x \neq \perp \ \& \ y \neq \perp \ \& \ x \in j_1(w_1(P_2)) \ \& \ y \in j_2(w_2(P_2))\} \cup \{\perp\}$
   $=w_\otimes(P_1)$
2. $i_\otimes(w_\otimes(P_1))$
   $=\{i_\otimes <x, y> \mid x \neq \perp \ \& \ y \neq \perp \ \& \ x \in w_1(P_1) \ \& \ y \in w_2(P_1)\} \cup \{i_\otimes(\perp)\}$
   $=\{<i_1(x), i_2(y)> \mid \ldots\} \cup \{\perp\}$
   $=\{<x, y> \mid x \neq \perp, y \neq \perp, x \in i_1(w_1(P_1)) \subseteq w_1(P_2) \ \& \ y \in i_2(w_2(P_1)) \subseteq w_2(P_2)\} \cup \{\perp\}$
   $\subseteq w_\otimes(P_2)$

Thus, $<i_\otimes, j_\otimes>: w_\otimes(P_1) \leftrightarrow w_\otimes(P_2)$ and the predicate transformer $w_\otimes$ can be used to construct a recursive predicate.

For $w_\oplus$:

1. $j_\oplus(w_\oplus(P_2))$
   $= \{<0, j_1(x)> \mid x \neq \perp \ \& \ x \in w_1(P_2)\} \cup \{<1, j_2(x)> \mid x \neq \perp \ \& \ x \in w_2(P_2)\} \cup \{\perp\}$

$= w_1(P_1) \oplus w_2(P_1)$

$= w_\oplus (P_1)$

2. $\iota_\oplus (w_\oplus(P_1))$

$= \{<0, \iota_1(x)> \mid x \neq \bot \;\&\; x \in w_1(P_1)\} \cup \{<1, \iota_2(x)> \mid x \neq \bot \;\&\; x \in w_2(P_1)\} \cup \{\bot\}$

$\subseteq w_1(P_2) \oplus w_2(P_2)$

$= w_\oplus(P_2)$

For $w_\rightarrow$:

1. $j_\rightarrow(w_\rightarrow(P_2))$

$= \{j_\rightarrow(f) \mid x \in w_1(P_2) \supset f(x) \in w_2(P_2)\}$

$= \{j_2 \circ f \circ \iota_1 \mid \ldots\}$

$= \{j_2 \circ f \circ \iota_1 \mid x \in w_1(P_1) \supset j_2(f(\iota_1(x))) \in w_2(P_1)\}$

$= \{F \mid x \in w_1(P_1) \supset F(x) \in w_2(P_1)\}$

$= w_\rightarrow(P_1)$

2. $\iota_\rightarrow(w_\rightarrow(P_1)$

$= \{ \iota_\rightarrow(f) \mid x \in w_1(P_1) \supset f(x) \in w_2(P_1)\}$

$= \{ \iota_2 \circ f \circ j_1 \mid x \in w_1(P_2) \supset \iota_2 (f(j_2(x))) \in \iota_2 (w_2(P_1)) \subseteq w_2(P_2)\}$

$\subseteq w_\rightarrow(P_2)$

If the predicate is defined as a tuple, a chain can be constructed as follows: $<\bot, \{<\bot, \bot>\}> \leq \ldots \leq <T^n(\bot), w^n<\bot, \bot>> \leq <T^{n+1}(\bot), w^{n+1}<\bot, \bot>> \leq \ldots$ such that $<\iota_n, j_n>: T^n(\bot) \leftrightarrow T^{n+1}(\bot)$, $<j_n, j_n> (w^{n+1}<\bot, \bot>) = w^n<\bot, \bot>$, and $<\iota_n, \iota_n> (w^n<\bot, \bot>) \subseteq w^{n+1}<\bot, \bot>$. Let $j_{2\otimes} = <j_n, j_n>$. Now, consider $w_{2\otimes}$:

1. $j_{2\otimes} (w_{2\otimes}(P_2))$

$=\{<j_\otimes, j_\otimes><<x, y>,<x', y'>> \mid$ none of the arguments are $\bot$ & $<x, x'> \in w_1(P_2)$ & $<y, y'> \in w_2(P_2)\} \cup \{<\bot, \bot>\}$

$= \{<<j_1(x), j_2(y)>, <j_1(x'), j_2(y')>> \mid \ldots\}$

$= \{<<a, b>, <a', b'>> \mid$ none of the arguments is $\bot$ & $<a, a'> \in j_1(w_1(P_2)) = w_1(P_1)$ & $<b, b'> \in j_2(w_2(P_2)) = w_2(P_1)\} \cup \{\bot\}$

$= w_{2\otimes} (P_1)$

2. $\iota_{2\otimes} (w_{2\otimes}(P_1)) = \{<\iota_\otimes, \iota_\otimes><<x, y><x', y'>> \mid \ldots\}$

$\subseteq w_{2\otimes}(P_2)$

There is a similar argument for $w_{2\oplus}$ and $w_{2\rightarrow}$.

Finally, consider $w_{1\rightarrow}$:

1. $j_{\rightarrow}(w_{1\rightarrow}(P_2))$
   $= \{j_{\rightarrow}(f) \mid <x, x'> \in w_1(P_2) \supset <f(x), f(x')> \in w_2(P_2)\}$
   $= \{j_2 \circ f \circ i_1 \mid \ldots\}$
   $= \{F \mid <x, x'> \in w_1(P_1) \supset <F(x), F(x')> \in w_2(P_1)\}$
   $= w_{1\rightarrow}(P_1)$
2. $i_{1\rightarrow}(w_{1\rightarrow}(P_1))$
   $= \{i_{\rightarrow}(f) \mid \ldots\}$
   $= \{i_2 \circ f \circ j_1 \mid \ldots\}$
   $= \{F \mid <x, x'> \in j_1(w_1(P_1)) = w_1(P_2)\ \&\ <F(x), F(x')> \in i_2(w_2(P_1)) \subseteq w_2(P_2)\}$
   $\subseteq w_{1\rightarrow}(P_2)$

Thus, the predicate transformers w can be used to construct structures for predicates, in particular, recursive predicates. The predicate transformers defined in the interpretation are simple because they are related to the domains in a very straightforward way. They allow the subsets and quotients of cpo's, which in turn, enable the definition of an interpretation for theories with domains.

6.7.2. Modelling the Quotient as a cpo

The quotient on the nonempty set U_A for domain A is defined as $U_A/U_{=} = \{U_S \mid U_S \subseteq U_A$ & for some $a \in U_A$, $U_S = \{a' \mid <a, a'> \in U_{=}\}\}$. Thus, the values in the quotiented carrier are equivalence classes. In Section 6.7 properties of the partial order for quotients are described. The partial order is derived from the order on the "unquotiented" domain. Let $x, y \in U_A$. Let $[x], [y] \in U_A/U_{=}$. The expression $a \in [x]$ means the value a is a member of the equivalence class [x]. The partial order is defined by: $[x] \leq [y]$ iff $(\forall a \in [x])(\exists b \in [y])\ a \leq b\ \&\ (\forall b \in [y])\ (\exists x \in [a])\ a \leq b$

A property of this order is:

$(x = y \ \& \ x \leq y) \supset x = y$

In the following sections we prove that the property holds and the order for quotients is a partial order. The least upper bound of a chain of equivalence classes is defined, and the domain isomorphisms specified in the source theory are discussed.

6.7.2.1. Partial Order Property 1

Property 1 states that $(x = y \ \& \ x \leq y)$ implies $x = y$. This means that elements in an equivalence class are unordered.

For flat cpo's only the bottom value is ordered with the other values. The formula $(\bot = x$ and $x \neq \bot)$ can only be true if = is not strict in both its arguments. The equivalence relation = is constructed from strict functions and the predicates &, V, ⊃, ¬, isl, isr, and =. Because coalesced products and sums of flat domains are flat domains, the predicates take a flat domain into another flat domain, bool. The usual truth-valued connectives &, v, and ⊃ have several monotonic extensions in $(bool \otimes bool \rightarrow bool)$. Select the ones that are strict in both their arguments. Similarly, select the strict extension of ¬. The predicates isl and isr are defined on $(A \oplus B \rightarrow bool)$, and = is defined on $(A \otimes A \rightarrow bool)$. If A is flat then there is a continuous test for equality such that equality is strict in both its arguments. Similarly, if A and B are flat then $A \oplus B$ is flat and define isl and isr to be strict in their argument. Assume any designer-specified functions and predicates are strict.

Thus, $=$ is strict in both its arguments and we have $\perp = x$ if and only if $x = \perp$; $\perp$ is not equivalent to any other value in a flat domain. From this, Property 1 is trivially true for flat domains.

Now proceed by induction on the domain construction to show that the property holds in general. First, consider non-recursive domains. Assume the property holds for $<U_A, \leq_A>$ and $<U_B, \leq_B>$. Consider the product domain $A \otimes B$. For property 1 we show $(<a, b> = <c, d> \;\&\; <a, b> \leq <c, d>) \supset <a, b> = <c, d>$ as follows:

$<a, b> = <c, d> \;\&\; <a, b> \leq <c, d>$

iff $a = c \;\&\; b = d \;\&\; a \leq c \;\&\; b \leq d$ (by definition of $=$ and $\leq$)

$\supset a = c \;\&\; b = d$ (by induction hypothesis)

iff $<a, b> = <c, d>$

Consider the sum domain $A \oplus B$. For property 1 we have:

$a = b \;\&\; a \leq b$

iff $(isl(a) \;\&\; isl(b) \supset outl(a) = outl(b) \;\&\; outl(a) \leq outl(b))$
$\&\; (isr(a) \;\&\; isr(b) \supset outr(a) = outr(b) \;\&\; outr(a) \leq outr(b))$
(by definition of $=$ and $\leq$)

$\supset (isl(a) \;\&\; isl(b) \supset outl(a) = outl(b)) \;\&\; (isr(a) \;\&\; isr(b) \supset outr(a) = outr(b))$
(by induction hypothesis)

iff $a = b$

Finally, consider the product domain $A \rightarrow B$. For property 1 we have:

$f = g \;\&\; f \leq g$

iff $(\forall a)(a = a' \supset f(a) = g(a')) \;\&\; f(a) \leq g(a)$

$\supset (\forall a)\, f(a) = g(a) \;\&\; f(a) \leq g(a)$

$\supset (\forall a)\, f(a) = g(a)$ (by induction hypothesis)

$\supset f = g$

The argument that the property holds for any domain D is similar to the argument above, but it relies on the inverse limit construction; thus, the notation is more tedious. We want to show that for any domain constructor T, $<x, y> \in M(=_{T(D)}) \cap M(\leq_{T(D)})$ implies $<x, y> \in M(=_{T(D)})$. If w is the predicate transformer corresponding to T then it is equivalent to say, $<x, y> \in w(M(=_D)) \cap w(M(\leq_D))$ implies $<x, y> \in w(M(=_D))$. The basis of the induction argument is for atomic domain D. The property holds because $M(=_D) \cap M(\leq_D)$ is $\{<\perp, \perp>\}$. Now, for $* \in \{\otimes, \oplus, \rightarrow\}$ and $T_*(D) = T_1(D) * T_2(D)$, we must show $<x, y> \in w_{2*}(M(=_D)) \cap w_{2*}(M(\leq_D))$ implies $<x, y> \in w_{2*}(M(=_D))$, assuming for n = 1, 2 that $<x, y> \in w_n(M(=_D)) \cap w_n(M(\leq_D))$ implies $<x, y> \in w_n(M(=_D))$. In the discussion below, we omit M in the expressions and assume the predicate symbols are the predicates.

Case 1:

$<x, y> \in w_{2\otimes}(=) \cap w_{2\otimes}(\leq)$

iff $<x, y> \in \{<a, b> \mid a \neq \perp$ & $b \neq \perp$ & $<pr1(a), pr1(b)> \in w_1(=)$
& $<pr2(a), pr2(b)> \in w_2(=)$
& $<pr1(a), pr1(b)> \in w_1(\leq)$ & $<pr2(a), pr2(b)> \in w_2(\leq)\} \cup \{\perp\}$

implies $<x, y> \in w_{2\otimes}(=)$, by induction hypothesis

Case 2:

$<a,<x,y>> \in w_{2\oplus}(=) \cap w_{2\oplus}(\leq)$

iff $<a, <x, y>> \in \{<b, <s, t>> \mid s \neq \perp$ & $t \neq \perp$
& (a = 0 implies $<s, t> \in w_1(=) \cap w_1(\leq)$) &
(a = 1 implies $<s,t> \in w_2(=) \cap w_2(\leq)$)$\} \cup \{\perp\}$

implies $<a, <x, y>> \in w_{2\oplus}(=)$, by induction hypothesis

Case 3:

$<x, y> \in w_{2\rightarrow}(=) \cap w_{2\rightarrow}(\leq)$

iff $<x, y> \in \{<f, g> \mid (<x, x'> \in w_1(=)$ implies $<f(x), g(x')> \in w_2(=))$
& $(<x, x'> \in w_1(\leq)$ implies $<f(x), g(x')> \in w_2(\leq))\}$

iff $<x, y> \in \{<f, g> \mid <x, x'> \in w_1(=) \cap w_1(\leq)$ implies $<f(x), f(x')> \in w_2(=) \cap w_2(\leq)\}$

implies $<x, y> \in w_{2\rightarrow}(=)$, by induction hypothesis.

6.7.2.2. Is the Defined Order Reflexive, Antisymmetric, and Transitive?

The order is reflexive because $[x] \le [x]$ follows from $[x] = [x]$. The order is antisymmetric because:

$$[x] \le [y] \;\&\; [y] \le [x]$$

$$\begin{aligned}
\text{iff } & (\forall x_1 \in [x])(\exists y_1 \in [y])\, x_1 \le y_1 \\
\& \ & (\forall y_2 \in [y])(\exists x_2 \in [x])\, x_2 \le y_2 \\
\& \ & (\forall y_3 \in [y])(\exists x_3 \in [x])\, y_3 \le x_3 \\
\& \ & (\forall x_4 \in [x])(\exists y_4 \in [y])\, y_4 \le x_4
\end{aligned}$$

$$\begin{aligned}
\text{iff } & (\forall x' \in [x])(\exists y_1\ \exists y_4)\, y_4 \le x' \le y_1 \\
\& \ & (\forall y' \in [y])(\exists x_2)(\exists x_3)\, x_2 \le y' \le x_3
\end{aligned}$$

$$\begin{aligned}
\text{iff } & (\forall x')(\exists y_1)(\exists y_4)\, y_4 = x' = y_1 \\
\& \ & (\forall y')(\exists x_2)(\exists x_3)\, x_2 = y' = x_3 && \text{(by property 1)}
\end{aligned}$$

$$\text{iff } (\forall x')(\forall y')\, [x'] = [y'] \qquad \text{(by property 1)}$$

$$\text{iff } [x] = [y]$$

The order is transitive because:

$$[x] \le [y] \;\&\; [y] \le [z]$$

$$\begin{aligned}
\text{iff } & (\forall x_1 \in [x])(\exists y_1 \in [y])\, x_1 \le y_1 \\
\& \ & (\forall y_2 \in [y])(\exists x_2 \in [x])\, x_2 \le y_2 \\
\& \ & (\forall y_3 \in [y])(\exists z_3 \in [z])\, y_3 \le z_3 \\
\& \ & (\forall z_4 \in [z])(\exists y_4 \in [y])\, y_4 \le z_4
\end{aligned}$$

$$\begin{aligned}
\text{iff } & (\forall x_1)(\exists z_3)\, x_1 \le z_3 \\
\& \ & (\forall z_4)(\exists x_2)\, x_2 \le z_4
\end{aligned}$$

$$\text{iff } [x] \le [z]$$

6.7.2.3. The Least Upper Bound of a Quotient Chain

In this section we prove that any chain of equivalence classes has a least upper bound (lub) by constructing the lub from the partial order definition and property 1. Basically, any chain of equivalence classes involves many chains connecting the elements of the equivalence classes. For the purposes of this discussion, a chain of equivalence classes (a chain in $U_D/U_{\simeq}$) is simply referred to as a quotient chain, and a chain connecting elements of an equivalence class (a chain in U_D) is referred to as a chain in the quotient chain. Below we show that the lub of a quotient chain is the equivalence class of the lub of any chain in the quotient chain.

Let D be a domain. Let $A = \{[a_1], [a_2], ...\}$ be a quotient chain in $U_D/U_{\simeq}$ such that $[a_i] \leq [a_{i+1}]$ for $i \geq 1$. First, we show that there is at least one chain in every quotient chain.

Lemma 3: For every $D/\simeq$-chain A there exists a D-chain $A'=\{a_1', a_2', ...\}$ such that $a_i' \simeq a_i$ for $i \geq 1$.

Proof: By the definition of $\leq$ we have $[a_i] \leq [a_{i+1}]$ iff $(\forall a_i' \in [a_i])(\exists a_{i+1}' \in [a_{i+1}])\ a_i' \leq a_{i+1}'$ & $(\forall a_{i+1}' \in [a_{i+1}])(\exists a_i' \in [a_i])\ a_i' \leq a_{i+1}'$. Pick any a_1' in $[a_1]$. Select an a_2' in $[a_2]$ such that $a_1' \leq a_2'$. We know this exists from the definition given above. By the same definition, there exists an a_3' in $[a_3]$ such that $a_1' \leq a_2' \leq a_3'$. Proceeding in this manner, we construct a chain $A' = \{a_1', a_2', ...\}$ such that $a_i' \simeq a_i$.

Next, we prove that the chain A′ constructed in the previous lemma is a maximal chain in A.

Lemma 4: The A′ chain constructed from A in the proof of Lemma 3 is a maximal chain in A.

Proof: Let $A' = \{a_1', a_2', ...\ a_i', a_{i+1}', ...\}$ be the constructed chain. A value cannot be added to A′ to get a longer chain in A. Assume we can add a value, say a_j', such that $a_i' \leq a_j' \leq a_{i+1}'$. Because $[a_i] \leq [a_{i+1}]$, we have $a_j' \simeq a_i$ or $a_j' \simeq a_{i+1}$. If we have $a_j' \simeq a_i$, then $a_i' = a_j'$, as a result of property 1. Similarly, if $a_j' \simeq a_{i+1}$ then $a_j' = a_{i+1}'$. Thus, A′ is a maximal chain in A.

Any chain in A proceeds in "lockstep" with any other chain in A.

Lemma 5: If $A' = \{a'_1, a'_2, \ldots\}$ and $A'' = \{a''_1, a''_2, \ldots\}$ are two chains in A, then for all i, $[a'_i] = [a''_i] = [a_i]$.

Proof: From Lemmas 3 and 4 we have $a'_i = a_i$ and $a''_i = a_i$. Therefore, $a'_i = a''_i$ and $[a'_i] = [a''_i] = [a_i]$.

Because (1) any chain in A proceeds in lockstep with any other chain in A and (2) there are no "dangling" chains (i.e., every element in every equivalence class is in at least one chain in the quotient chain A), the lub of the quotient chain can be constructed from the lub of any chain in the quotient chain.

Theorem 6: For D/=-chain A, lub(A) = [lub(A')] where A' is any D-chain in A.

Proof: By Lemmas 3 and 4, there exists a D-chain in A, call it A', and this chain is maximal. Because D is a domain, there exists lub(A'). To show lub(A) is well defined, assume A" is another maximal D-chain in A. We have to show [lub(A')] = [lub(A")]. By Lemma 5, we have $a'_i = a''_i$ for all i. Because = is inclusive, we have lub(A') = lub(A").

6.7.3. Does the Derived Source Structure Model Isomorphisms Specified Among Source Domains?

If D = F is a domain equation in the source theory then we must show that $<U_D, \leq_D>$ is isomorphic to $<U_F, \leq_F>$ where the cpo's are derived from the target theory. Consider the following three source domain equations:

1. A = D ⊗ E
2. B = D ⊕ E
3. C = D → E

The carriers for these domains are defined as follows:

1. $U_A = \{U_x \mid x \subseteq \text{is-}(D \otimes E)\ \&\ \text{for some } <d, e> \in U_{\text{is-}(D\otimes E)}, U_x = \{<d', e'> \mid <d', e'> =_{D\otimes E} <d, e>\}\}$
2. $U_D \otimes U_E = \{<U_y, U_z> \mid y \subseteq \text{is-}D\ \&\ z \subseteq \text{is-}E\ \&\ \text{for some } d \in U_{\text{is-}D}, U_y = \{d' \mid d' =_D d\}\ \&\ \text{for some } e \in U_{\text{is-}E}, U_z = \{e' \mid e' =_E e\}\}$

3. $U_B = \{U_x \mid x \subseteq \text{is-}(D\oplus E)\ \&\ \text{for some } v \in U_{\text{is-}(D\oplus E)},\ U_x=\{v' \mid v' =_{D\oplus E} v\}\}$
4. $U_D \oplus U_E = \{<0, U_y> \mid y \subseteq \text{is-}D\ \&\ \text{for some } d \in U_{\text{is-}D},\ U_y= \{d' \mid d' =_D d\}\} \cup$
   $\{<1, U_z> \mid z \subseteq \text{is-}E\ \&\ \text{for some } e \in U_{\text{is-}E},\ U_z = \{e' \mid e' =_E e\}\} \cup \{\bot\}$
5. $U_c = \{U_x \mid x \subseteq \text{is-}(D\rightarrow E)\ \&\ \text{for some } f \in U_{\text{is-}(D\rightarrow E)},\ U_x = \{f' \mid f' =_{D\rightarrow E} f\}\}$
6. $U_D \rightarrow U_E = \{F \mid$ for some $f \in U_{\text{is-}(D\rightarrow E)}$, F is a continuous function from $\{U_y \mid y \subseteq \text{is-}D\ \&\ \text{for some } d \in U_{\text{is-}D},\ U_y = \{d' \mid d' =_D d\}\}$ to $\{U_z \mid z \subseteq \text{is-}E\ \&\ \text{for some } e \in f[U_{\text{is-}D}],\ U_z = \{e' \mid e' =_E e\}\}$

To show that the source domain equations are satisfied, isomorphisms Θ_1, Θ_2, Θ_3 are defined such that:

1. $\Theta_1: U_A \rightarrow (U_D \otimes U_E)\ \&\ x \leq_A y \text{ iff } \Theta_1(x) \leq_{D\otimes E} \Theta_1(y)$
2. $\Theta_2: U_B \rightarrow (U_D \oplus U_E)\ \&\ x \leq_B y \text{ iff } \Theta_2(x) \leq_{D\oplus E} \Theta_2(y)$
3. $\Theta_3: U_C \rightarrow (U_D \rightarrow U_E)\ \&\ x \leq_C y \text{ iff } \Theta_3(x) \leq_{D\rightarrow E} \Theta_3(y)$

The isomorphisms are defined as follows:

1. $\Theta_1(\{<d', e'> \mid <d', e'> =_{D\otimes E} <d, e>\}) = <\{d' \mid d' =_D d\}, \{e' \mid e' =_E e\}>$
2. $\Theta_2(\{<0, d'> \mid <0, d'> =_{D\oplus E} <0, d>\}) = <0, \{d' \mid d' =_D d\}>$, and similarly for inr
3. $\Theta_3(\{f' \mid f' =_{D\rightarrow E} f\}) = F$, where $f(a)= b \supset F([a])=[b]$.

Now, Θ_1 and Θ_2 are obviously isomorphisms (1-1 and onto). Θ_3 is discussed below. Let $\Theta_3([f])=F$ and $\Theta_3([g])=G$. The claim that Θ_3 is well-defined is shown in the following two steps:

1. $f=g$
   $\supset a=a' \supset f(a)= g(a')$
   $\supset [a] = [a'] \supset [f(a)] = [g(a')]$
   $\supset [a] = [a'] \supset F([a]) = G([a'])$
   $\supset F=G$
   $\supset \Theta_3([f]) = \Theta_3([g])$
2. $a=a'$
   $\supset f(a) = f(a')$ (because $f \in U_{\text{is-}(D\rightarrow E)}$)
   $\supset [f(a)] = [f(a')]$
   $\supset F([a]) = F([a'])$
   $\supset \Theta_3([f])([a]) = \Theta_3([f])([a'])$

To show Θ_3 is 1-1 we must show $[f] \neq [g] \supset \Theta_3([f]) \neq \Theta_3([g])$ where $[f] = \{f' \mid f' = f\}$.

We have:

$$
\begin{aligned}
&[f] \neq [g] \\
&\supset \neg(f = g) \\
&\supset (\exists a)\ \neg(f(a) = g(a)) \\
&\supset (\exists a)\ [f(a)] \neq [g(a)] \\
&\supset (\exists [a])\ F([a]) \neq G([a]) \\
&\supset F \neq G
\end{aligned}
$$

To show Θ_3 is onto we must show $(\forall F)(\exists g)\ \Theta_3([g]) = F$. The function F is in $[U_{is\text{-}D}/U_{=_D} \rightarrow f(U_{is\text{-}D})/U_{=_E}$ for some $f \in U_{is\text{-}(D \rightarrow E)}$. Let g be that particular f. The function f is a continuous function such that

1. $is\text{-}D(x) \supset is\text{-}E\ (f(x))$
2. $x =_D x' \supset f(x) =_E f(x')$

If $F([a]) = [b]$ then $f(a) = b$. Furthermore, $a =_D a' \supset b = f(a')$. By definition of Θ_3 and f we have:

$$
\begin{aligned}
&\Theta_3([f])([a]) \\
&= [f(a)] \\
&= [b]
\end{aligned}
$$

Therefore, $\Theta_3([f]) = F$.

The property $[<a,b>] \leq_A [<c,d>]$ iff $\Theta_1([<a, b>]) \leq_{D \otimes E} \Theta_1([<c, d>])$ holds by the following argument:

$$
\begin{aligned}
&[<a, b>] \leq [<c, d>] \\
&\text{iff } (\forall <a', b'> \in [<a, b>])\ (\exists <c', d'> \in [<c, d>]\ <a', b'> \leq <c', d'>\ \& \\
&(\forall <c', d'> \in [<c, d>])\ (\exists <a', b'> \in [<a, b>])\ <a', b'> \leq <c', d'> \\
&\text{iff } (\forall a' \in [a])\ (\exists c' \in [c])\, a' \leq c' \\
&\&\ (\forall c' \in [c])\ (\exists a' \in [a])\ a' \leq c' \\
&\&\ (\forall b' \in [b])\ (\exists d' \in [d])\ b' \leq d' \\
&\&\ (\forall d' \in [d])\ (\exists b' \in [b])\ b' \leq d' \\
&\text{iff } <[a], [b]> \leq <[c], [d]>
\end{aligned}
$$

The argument for the corresponding property for sums is similar.

The property $[f] \leq_c [g]$ iff $\Theta_3([f]) \leq_{D \to E} \Theta_3([g])$ holds by the following argument:

$[f] \leq [g]$

iff $(\forall a)(\forall f' \in [f])(\exists g' \in [g])\ f'(a) \leq g'(a)\ \&\ (\forall g' \in [g])(\exists f' \in [f])\ f'(a) \leq g'(a)$

iff $(\forall a)\ [f(a)] \leq [g(a)]$

iff $(\forall a)\ \Theta_3\ ([f])([a]) \leq \Theta_3\ ([g])([a])$

iff $\Theta_3\ ([f]) \leq \Theta_3\ ([g])$

What all this means is that the domain equations specified in the source theory are true in the source structure, where the source structure is derived from the target theory structure and the interpretation. With the interpretation presented in this paper, the designer does not have to prove that the interpreted domain isomorphisms hold in the target theory because the domain interpretations are constructed in a manner that preserves this property.

6.7.4. Deriving Source States

Scott showed in [scott 76] how to model everything in one "universal" domain, the domain of all subsets of the set of nonnegative integers. If U is a universal domain, then every domain D is isomorphic to a subdomain of U. In particular, $U \to U$, $U \otimes U$, and $U \oplus U$ are all isomorphic to subdomains of U. It is possible to view $x \in U$ at one time as a value, at another as an argument to a function, then as an integer, and later as a function.

Similarly, the derivation of source domain structures can be achieved in different ways depending on whether it is desirable to view domain values as single arguments, or as structured arguments. Briefly, the mapping J from target states to source states is defined as $J\rho x = J_D(M(I(x))(\rho))$, where variable symbol x has signature D, and source state $(J\rho)$ assigns a value to x. If D is an atomic source domain, then x can only be used as an argument and $J_D(M(I(x))(\rho) = [M(I(x))\rho]$; a value in D's structure is some equivalence class. If D is a function space, say A→B, then a value in D's structure can be viewed

1. as some equivalence class of functions, $[M(I(x))\rho] \in U_{is\text{-}(A\to B)}/U_{=_{A\to B}}$ where $J_{A\to B}(M(I(x))\rho) = [M(I(x))\rho]$, or
2. as some function that takes an equivalence class as an argument and returns an equivalence class as a result, $F \in (U_{is\text{-}A}/U_{=_A} \to (M(I(x))\rho)(U_{is\text{-}A})/U_{=_B})$ where $F([a]) = [(M(I(x))\rho)(a)]$.

Say g: A → B → C is in the source theory. Then, $M(g(x))(J\rho) = (M(g)(J\rho))\ ([M(I(x))\rho]) = [(M(I(g))\ (\rho)\ (M(I(x))\rho)]$, while $M(x(a))(J\rho) = F([M(I(a))\rho]) = [(M(I(x))\rho)\ (M(I(a))\rho)]$.

If D is a product, say A ⊗ B, then a value in D's structure can be viewed.

1. as one argument, $J_{A\otimes B}\ (M(I(x))\rho) \in U_{is\text{-}(A\otimes B)}/U_{=_{A\otimes B}}$, or
2. as two arguments (an argument pair), $\langle J_A(pr1 \circ M(I(x))\rho),\ J_B(pr2 \circ M(I(x))\rho)\rangle \in (U_{is\text{-}A}/U_{=_A} \otimes U_{is\text{-}B}/U_{=_B})$

Note that $\theta_1(J_{A\otimes B}\ (\langle a, b\rangle)) = \langle J_A\ (a), J_B\ (b)\rangle$. Say, h: A⊗B → C, $d \in U_{is\text{-}A\otimes B}$, $a \in U_{is\text{-}A}$, and $b \in U_{is\text{-}B}$. Then the meaning of h can be a function that maps [d] to $[(M(I(h))\rho)(d)]$, or a function that maps $\langle [a], [b]\rangle$ to $[(M(I(h))\rho)(\langle a, b\rangle)]$.

Finally consider D a sum, say A ⊕ B. Then a value in D's structure can be viewed

1. as an argument that is not identified as belonging to one of the summands, $J_{A\oplus B}(M(I(x))(\rho) \in U_{is\text{-}(A\oplus B)}/U_{=_{A\oplus B}}$, or
2. as an argument that is identified as belonging to one of the summands, $\langle 0, J_A(a)\rangle$ if $M(I(x))\rho = \langle 0, a\rangle$ and $\langle 1, J_B(b)\rangle$ if $M(I(x))\rho = \langle 1, b\rangle$. The argument is in $U_{is\text{-}A}/U_{=_A} \oplus U_{is\text{-}B}/U_{=_B}$.

Note that $\theta_2(J_{A\oplus B}(<0, a>)) = <0, J_A(a)>$. Say k: A⊕B → C, $d \in U_{is\text{-}(A\oplus B)}$, and y: A. Then, the meaning of k is a function that maps [d] to [(M(I(k))ρ)(d)], and the meaning of k(inl(y)) is [(M(I(k))ρ)(<0, [M(I(y))ρ]>)].

6.7.5. Are the Derived Source Operations Continuous?

Let D→E be a source domain. Because both $U_{is\text{-}(D\to E)}/U_{\equiv_{D\to E}}$ and $(U_{is\text{-}D}/U_{\equiv_D} \to f(U_{is\text{-}D})/U_{\equiv_E})$ for $f \in U_{is\text{-}(D\to E)}$ are cpo's, any value in them is a continuous function. Thus, the function assigned to h: D→E is continuous. Another way to look at it is to see how h is implemented. Assuming the target operations are continuous it can be shown that the derived source structure assigns continuous functions to source function symbols. Let h^S be the operation the source structure assigns to h where h^S: $U^S_D \to U^S_E$: $[d] \to [I(h)^T(d)]$. This follows from the fact that we have is-(D→E)(I(h)). Therefore, for chain $A=\{[a_1], [a_2], \ldots\}$, $h^S(A) = \{[I(h)^T(a_1)], [I(h)^T(a_2)], \ldots\}$. By the monotonicity of $I(h)^T$, if $a_i \le a_{i+1}$ then $I(h)^T(a_i) \le I(h)^T(a_{i+1})$.

Assume $a_i \le a_{i+1}$ for all i. Let $A'=\{a_1, a_2, \ldots\}$. Then

$h^S(\text{lub}(A))$

$= h^S([\text{lub}(A')])$ (by partial order definition for quotients)

$= [I(h)^T(\text{lub}(A'))]$ (definition of h^S)

$= [\text{lub}(I(h)^T(A'))]$ (by continuity of $I(h)^T$)

$= \text{lub}([I(h)^T(A')])$ (by partial order definition for quotients)

$= \text{lub}\ (h^S\ (A))$ (definition of h^S)

Thus, h^S is continuous.

6.8. Wand's Theorem 4.1 Revisited

Assuming the predicates $=_D$ and is-D exist, we show that the correctness conditions are sufficient. Consider the following propositions:

Proposition 7: If the interpretation I satisfies the correctness criteria then $I(=_D)$ is an equivalence relation for any source domain D.

Proof: By correctness conditions 3 and 4, $I(=_D)$ is an equivalence relation for any atomic source domain D. Denote $I(=_D)$ as $=_D$. Assume $=_A$ and $=_B$ are equivalence relations. Then $=_{A \otimes B}$ is an equivalence relation because reflexivity, transitivity, and symmetry follow from the definition of $=_{A \otimes B}$. Similarly, for $=_{A \oplus B}$. For $=_{A \to B}$, where f, g, and h are in is-(A→B) we have

1. $(a =_A a' \supset fa =_B fa') \supset f =_{A \to B} f$
2. $f =_{A \to B} g \supset (a =_A a' \supset fa =_B ga') \supset (a' =_A a \supset ga' =_B fa) \supset g =_{A \to B} f$
3. $(f =_{A \to B} g \;\&\; g =_{A \to B} h) \supset (a = a' \supset fa = ga' = ga = ha') \supset f =_{A \to B} h$

Note that for recursive domains D, $=_D$ exists and is inclusive. Therefore, for chains X and Y in D, if X = Y then lub(X) = lub(Y). So, = is reflexive, transitive and symmetric for chains. It follows that the solution to any definition for = is an equivalence relation.

Proposition 8: If the interpretation I satisfies the correctness criteria then $T_{target} \vdash (\exists x)$ is-D(x) for any source domain D.

Proof: By correctness condition 1, $T_{target} \vdash (\exists x)$ is-D(x) for any atomic source domain D. Assume is-A and is-B define nonempty sets in the target. Then is-(A⊗B) and is-(A⊕B) define nonempty sets. Also, the set x = {x | is-A(a) ⊃ is-B(x(a))} is nonempty. Now, does there exist $x \in X$ such that $a=a' \supset x(a) = x(a')$? Define x such that $(\forall a)\ x(a)=b$ for some b. Thus, is-(A→B) defines a nonempty set.

If D is recursively defined, its solution could be ⊥. In this case the is-D subset is nonempty because the subset contains ⊥. More generally, is-D always exists and is inclusive. Therefore, maximal chains, which represent a sequence of approximations, are in the subset and it is nonempty.

We have by Propositions 7 and 8 that $=_D$ can be used to define a quotient domain and is-D is non-empty. Thus, analogous to [wand 82a], a carrier for a source domain is the partitioned subset of the carrier for the source domain interpretation. However, in this research a domain carrier is defined as a cpo, a set that has additional properties.

Because the definition of J is the same (modulo the bottom element) as in [wand 82a] and the introduction of domains does not alter the grammatical structure of formulas, the proof by structural induction of Theorem 4.1 is basically the same as that for a many-sorted first-order theory. The differences were accounted for in the previous sections where we showed that the map J did indeed produce a source structure, even though we used cpo's and continuous functions instead of sets and total functions.

6.9. Simplification of Correctness Proofs

There are several obvious things one can do to eliminate some "clutter" in interpreted formulas and to eliminate some of the work needed to verify the correctness criteria. These simplifications arise when source objects do not change their representation in the implementation in any significant manner. For example, a projection operator for a product domain (e.g., pr1) will be represented by a projection operator. Even though the source product domain is represented by a target product domain where the constituent domains of the target product may differ from those of the source, the same axioms will specify the projection operator in both the source and the target. In this case, the projection operator, say pr1, can be removed from the list of free symbols and thus, is not incorporated into the preamble of a formula that refers to pr1.

Also the interpreted axioms specifying pr1 are trivially true in the target theory. Consider the following propositions:

Proposition 9: Indentity Map Theorem If h: $D_1 \rightarrow D_2$ is a fixed operator symbol in the theory schema and I(h) = h, then is-$(D_1 \rightarrow D_2)$(h) = TRUE.

Proof: If h ∈ {¬, ⊃, V, &, cond, pair, pr1, pr2, outl, outr, inl, inr, isl, isr, id, °, curry, uncurry, TRUE, FALSE} then I(h) = h. Consider the case where I(pr1: A⊗B→A) = pr1: I(A)⊗I(B)→I(A). Denote the projection operator in the target as pr1′. We have

$\text{is-}(A \otimes B \rightarrow A)(pr1')$

iff $(\text{is-}A(a) \;\&\; \text{is-}B(b) \supset \text{is-}A\,(pr1'(<a, b>)))$
$\&\; (a =_A a' \;\&\; b =_B b' \supset pr1'(<a, b>) =_A pr1'\,(<a', b'>)$

iff TRUE, because $pr1'(<a, b>) = a$ and $pr1'(<a', b'>) = a'$

For the pair operator,

$\text{is-}(A \rightarrow B \rightarrow (A \otimes B))(pair)$

iff $(\text{is-}A(a) \supset (\text{is-}B(b) \supset \text{is-}(A \otimes B)(pair(a,b))))$
$\&\; (a =_A a' \supset (b =_B b' \supset pair(a,b) =_{A \otimes B} pair(a', b')))$

iff TRUE, because $pr1(<a, b>) = a$, etc.

For outl,

$\text{is-}(A \oplus B \rightarrow A)(outl)$

iff $(\text{is-}(A \oplus B)\,(c) \supset \text{is-}A(outl(c)))$
$\&\; (c =_{A \oplus B} c' \supset outl(c) =_A outl(c'))$

iff TRUE,
because $isr(c) \supset M(outl(c)) = \bot$, $M(\text{is-}A)(\bot) = TRUE$,
and $M(=_A)\,(\bot, \bot) = TRUE$

For inl,

$\text{is-}(A \rightarrow A \oplus B)(inl)$

iff $(\text{is-}A(a) \supset \text{is-}(A \oplus B)(inl(a))) \;\&\; (a =_A a' \supset inl(a) =_{A \oplus B} inl(a')$

iff TRUE

For isl,

$\text{is-}(A \oplus B \rightarrow bool)(isl)$

iff $(\text{is-}(A \oplus B)(c) \supset TRUE) \;\&\; (c =_{A \oplus B} c' \supset isl(c) = isl(c'))$

iff $((isl(c) \;\&\; isl(c') \supset outl\,(c) =_A outl(c')) \;\&$
$(isr(c) \;\&\; isr(c') \supset outr(c) =_B outr\,(c'))) \supset$
$isl(c) = isl(c'))$

iff TRUE, because $isl(c) \neq isl(c')$ iff $(isl(c) \;\&\; isr(c'))$
or $(isr(c) \;\&\; isl(c'))$

For id, is-(D→D)(id) iff (is-D(d) ⊃ is-D(id(d)) and d $=_D$ d′ ⊃ id(d) $=_D$ id(d′) which is obviously true because id(d) = d.

For the composition operator °,

is-((A→B)⊗(B→C)→(A→C))(°)

iff (is-(A→B)(f) & is-(B→C)(g) ⊃ is(A→C)(g°f)) & (f $=_{A\to B}$ f′ & g $=_{A\to B}$ g′ ⊃ g ° f $=_{A\to C}$ g′ ° f′)

iff ((is-A(a) ⊃ is-B(f(a))) & (is-B(b) ⊃ is-C(g(b)))
  ⊃ is-A(a) ⊃ is-C(g ° f(a)))
& ((a = a′ ⊃ f(a) = f(a′)) & (b = b′ ⊃ g(b) = g(b′))
  ⊃ (a = a′ ⊃ g ° f(a) = g ° f(a′)))
&((a = a′ ⊃ f(a) = f′(a′)) & (b = b′ ⊃ g(b) = g′(b′))
  ⊃ (a = a′ ⊃ g ° f(a) = g ′ ° f′(a))

iff TRUE, because g ° f(a) = g(f(a))

For curry,

is-((A⊗B→C)→(A→B→C))(curry)

iff is-(A⊗B→C)(f) ⊃ is-(A→B→C)(curry(f))
& f $=_{A\otimes B\to C}$ f′ ⊃ curry(f) $=_{A\to B\to C}$ curry(f′)

iff((is-(A⊗B)<a,b> ⊃ is-C(f(<a,b>)))
  ⊃ (is-A(a) ⊃ is-B(b) ⊃ is-C(curry(f)(a)(b))))
& ((<a,b> = <a′, b′> ⊃ f(<a, b>) = f(<a′ b′>))
  ⊃ (a = a′ ⊃ b = b′ ⊃ curry(f)(a)(b) = curry (f)(a′)(b′)))
& ((<a, b> = <a′, b′> ⊃ f(<a, b>) = f′(<a′, b′>))
  ⊃ (a = a′ ⊃ b = b′ ⊃
  curry(f)(a)(b) = curry(f′)(a′)(b′)))

iff TRUE, because curry(f)(a)(b) = f(<a, b>)

The other operators are similar.

Proposition 10: Preamble Simplification Theorem. If, as described in the Identity Map Theorem, I(h) = h, then expression is-(D_1 → D_2)(h) can be eliminated from any preamble.

Proposition 11: Criteria Simplification Theorem The interpretation of theory schema axioms specifying products and sums are (trivially) deducible in the target theory.

Of course, the designer may also specify is-D(d) = TRUE for atomic domain D. This occurs when any value in the target domain I(D) is a legal source representative.

Chapter 7

Application of Interpretation Between Theories to the Compiler Design Correctness Problem

7.1. Correctness Criteria - Chapter Overview

This chapter illustrates how the theories and interpretations are specified for the compiler design correctness proof and discusses the proof process. The specification of a programming language as an abstract data type is discussed. Denotational semantics is selected for specifying programming language semantics and is incorporated into the theory specifying the programming language. Assuming the specification language is based on denotational semantics, a compiler design is defined as an interpretation of L_{source} to L_{target}. The algorithms for translating axioms in T_{source} and strategies for deducing the translated axioms in T_{target} are discussed.

7.2. Defining Programming Languages as Higher Order Abstract Data Types

An abstract data type is a set of operations and the definitions of the relationships between the operations. We take the position as in [wand 80] that a programming language is, semantically, just a complex data type (or conversely, a data type is just a simple programming language). A programming language specification can be defined as an abstract data type where there are operations for

1. building program phrases

2. assigning meanings to program phrases

The two groups of operations are called the *defined language* and the *defining language*, respectively. This terminology is used in [reynolds 72]. Both languages constitute the language for the theory that defines a programming language.

The defined language is based on the context free grammar of the programming language. For example, the following production, written in BNF, defines the structure of a command:

command → identifier := expression |
output expression |
if expression **then** command **else** command

This is converted to the following three function symbols and their signatures:

:=: identifier ⊗ expression → command
output: expression → command
if: expression ⊗ command ⊗ command → command

The domain symbols in the signatures correspond to the non-terminal symbols in the BNF rule; they identify the type of a syntactic object. The defined language is actually the "abstract syntax" of the programming language. The abstract syntax defines the structure of a phrase in terms of constituent phrases. The syntactic sugaring in the BNF rule (e.g., **then**) was removed by converting the programming language to prefix notation. With abstract syntax it is clear how to construct a syntactic object, but not how to write it. At this point, parsing is not considered in the proof.

The defining language contains operations which evaluate the defined language; i.e., it is the semantics of the programming language. We regard the meaning of a program phrase to be a mathematical object. The operations in the defining language are functions which take elements of a defined language sort as arguments and return elements of a defining language sort. The functions in the defining language are a homomorphism from the defined language to lambda calculus (or combinator calculus). This means the defining language is a denotational semantics for the defined language. The denotational semantics is specified as a formal system. A major difference between the language of the formal system here and the languages described in Appendices A and B is that the language has higher order operations, domains, and domain equations.

The operators in the defining language are taken to be the semantic functions associated with the denotational semantics of the programming language. The axioms and/or rules of inference specify the semantic equations for the programming language. Denotational semantics was selected because the method applies to a wide variety of programming constructs, including most of those in Algol 60, Pascal, and LISP.

An example of the defining language is the command continuation domain, cont, specified as cont = state → answer. There would be a semantic operator, such as C: command → cont → cont where the meaning of a command, an element of a syntactic domain, is an element of the function domain (cont → cont), a semantic domain. If command was specified as above, then there would be three axioms, each specifying the behavior of a particular command in terms of the semantic operator C.

The additional operator symbols in the theory used to define functional application and abstraction depend on whether terms are written in lambda calculus or in combinator calculus. Combinator calculus has the same meaning as lambda calculus; they are two different notations for a functional high-level language. Both may be considered because they have different effects on the efficiency of the translation and deduction necessary for the correctness proof.

If lambda calculus is used, there are operators for expressing lambda abstraction and operators for lambda applications. The combinator calculus also has application operators. The combinators in combinator calculus are additional constants that are defined as lambda expressions. Lambda expressions can be translated to combinator expressions where the translation produces an expression without bound variables. Some researchers are suggesting that because lambda expressions are easier to read the specifications should be written in lambda calculus and the combinator calculus used internally in an automated verification system [turner 79]. However, there is much work to be done on this issue and future research in this area is proposed. For purposes of readability, lambda calculus is used in this research. Examples of $\dot{T}_{source}$ and T_{target} are presented in Chapter 8.

7.3. Specifying the Compiler Design as an Interpretation

Because the defined language is used to specify abstract syntax and the defining language is used to specify semantics, the compiler design is an interpretation that maps the source defined language into the target defined language (syntactic domains to syntactic domains) and maps the source defining language to the target defining language (semantic domains to semantic domains). For example, a source syntactic domain **expression** can be interpreted as a target syntactic domain **code**, where, in particular, the source expression constant **1** is interpreted as the machine instruction **[loadn, 1]**. Examples for the defining language include interpreting a memory domain as a memory domain, or interpreting various source continuations, such as a command continuation, an expression continuation, or a declaration continuation, as some machine continuation. At a more detailed level of design, perhaps numbers are interpreted as bitstrings, and stacks as memory-counter pairs. Detailed examples are presented in Chapter 8.

However, as it was noted earlier, the source and target semantics must be written in the same style in order to find an interpretation, as defined in this paper. This enables one to construct a correctness proof based implicitly on structural induction on the source language. This is explained in the next section. As we will explain later, this has the same applicability as the algebraic approach to compiler design correctness, but results in a different proof organization. Other verification methods will briefly be discussed. They result in complicated induction arguments and may be difficult to apply in large-scale problems.

A primary concern is that the verification process should mirror the informal specification and justification that is actually done by a designer. The verification process should be a natural extension to the design and provide a reasonable document of the work done. The compiler designer maps each source programming language construct into some target code and does a mental comparison of the source construct behavior and the construct translation behavior. This is typically done independent of the other constructs. Perhaps the designer perceives the source construct in a certain state in an arbitrary program, and mentally views the relation between input to and output from the construct, including any possible side effects it has. The source construct translation is perceived in the implemented state in an arbitrary machine program. This has some input/output behavior and side effects. With assumptions about states and program surroundings, the mental comparison of behavior is done. With this in mind, it seems fairly natural to apply the approach proposed in this paper. It does not seem natural that the designer mentally constructs elaborate machines that interpret each programming language and then determines how any state in one machine is implemented in the other machine.

Programming language semantics are used in this application to determine the effect of a representation change, and similarity of source and target semantic styles enables a straightforward analysis of the representation change. If the programming language specifications are developed *a priori*, then it may be impossible to find an interpretation. However, if this is the case, the source and target specifications may be rewritten so that they have the same semantic style, and other methods used to show that specifications written in different semantic styles define the same programming language. This is an easier problem because one would just have to focus on the change of semantic domain as there would be no representation change of the programming language syntax. Also, one could refer to publications for examples of how to rewrite, say, a direct style specification as a continuation style specification, or a store style as a state style.

Lastly, some work, such as [wand 82b] and [royer 86] along with compiler-compiler research is being done where, rather than given source and target specifications independently, the target specification is derived from the source using semantic preserving manipulations. In this work, the derived target semantics is similar to the source semantics, but one step closer to an implementation. Hence, the applicability of the verification approach proposed in this paper to verify the correctness of the derivations. In fact, it is most likely the proposed verification approach would succeed for proving derived target semantics correct or for verifying the compilation of a source language into some "intermediate" language. This research may help provide a means of certifying a multi-level design. Some traditional certification methods require refinement to the lowest level.

7.4. Correctness Proof Based on Structural Induction

In proving the correctness of a compiler, it must be shown that the compiler is correct for *any* arbitrary input to the compiler. The traditional method of debugging demonstrates that a compiler will only work for some sample input. "To prove that it works for arbitrarily complex data it is natural to define data objects inductively. We then show that it works for the most elementary data, and that it will work for data of any degree of complexity provided that it works for all data of lesser complexity. We may then induce that it works for all data [burstall 68]." This method of proof is called *structural induction*.

The inductive ordering is defined in terms of the relation "constituent". An object A is a constituent of object B if A is identical with B or if A is a constituent of a component of B. A proper constituent is a constituent of an object that is not identical to the object. The induction principle for this ordering is: if for some set of structures a structure has a certain property whenever all its proper constituents have that property then all the structures in the set have the property.

If the compiler is not optimized or optimization occurs after the target code is produced, the compilation of each syntactic type is independent of the compilation of other syntactic types. Thus, compiler correctness can be stated in terms of the compilation correctness of each syntactic type. Structural induction is used to prove more complex source language syntactic cases correct in terms of syntactic objects of lesser complexity.

The inductively defined data object in the compiler design correctness proof is the source theory. The abstract syntax of the source language specifies each syntactic type of the source language in terms of constituent syntactic types. The denotational semantics of each syntactic type is defined in terms of the semantics of the constituent syntactic types. All legal program phrases and true properties about program phrases are deduced from the theory. If all objects in the source theory are correctly implemented, then any source program is correctly implemented. This is stated formally in the Implementation Theorem; if the interpretation is correct then the implementation of anything deducible in the source theory is deducible in the target theory.

The Implementation Theorem was proved by structural induction where the inductive ordering is on the grammar of the theory. Thus, the structural induction foundation is established once, and the designer can ignore the details and follow the recipe given in the correctness criteria. The induction argument is automatically incorporated into the mapping that occurs when the interpretation is applied. This is a mechanical process. After the interpretation is applied, the correctness proof proceeds by deduction in the target theory, again, much of which is a mechanical process. The proof primarily involves rewriting terms using the semantic equations in the target theory.

Thus, two possible advantages can be achieved by using the proposed verification method. One, the induction argument, which in some methods is interleaved throughout the proof obscuring the argument and making mechanization difficult, is achieved simply and painlessly as a translation. Two, the proof is done in a relatively small environment, the target theory -- some other methods require simplification using both the source and target theories.

Chapter 8

Examples

8.1. Stacks - Example of Subsets and Quotients in the Interpretation

Wand gives a good example of how stacks are implemented by array-integer pairs in [wand 82a]. This is reviewed in Appendix B. It illustrates how and why the predicates is-stk and $\approx_{stk}$ are defined. If the integer represents the top of the stack and the stack contents are represented by array contents from location one to the positive integer value, then an array-integer pair is a stack representative if the integer is greater than or equal to zero. Two array-integer pairs are stack equivalent if their integer parts are equal and when the integer is greater than zero, their array contents from one to the integer value are equal. One can imagine the usefulness of subsets and quotients in a computer application because one can imagine specifying memory components as arrays and situations where it would be desirable to view different memory configurations as equivalent and certain memory configurations as illegal.

This example also points out the dangers of overspecification. If the source theory, the theory of stacks, is overspecified, it may restrict or prevent various implementations, or lead to inefficient and unnatural implementations. Consider the following five (incomplete) specifications of a stack. The first three define unbounded stacks and the last two define stacks with maximum length of 100.

1. (unbounded, atomic stack spec.) atomic domain stk_1 and axioms, such as pop(push(s, v)) = s

2. (unbounded, finite length stack spec.) domain equation $stk_2 = val^*$ where val^* is $1 \oplus val \oplus (val \otimes val) \oplus (val \otimes (val \otimes val)) \oplus \ldots$
3. (finite and infinite length stack spec.) domain equation $stk_3 = 1 \oplus (val \otimes stk_3)$
4. (bounded, atomic stack spec.) atomic domain stk_4 and axioms, such as $length(s) < 100 \supset pop\ (push(s, v)) = s$
5. (bounded stack spec.) domain equation $stk_5 = 1 \oplus val \oplus val^2 \oplus \ldots \oplus val^{100}$

Specifications 2 and 5 have the concept that a stack "carries around" its length. In specification 4, the length can be calculated when necessary. Now, consider four (incomplete) specifications of an array, two unbounded and two bounded.

1. (unbounded, atomic array spec.) atomic domain arr_1 and axioms, such as $retrieve(store(a, i, v)) = v$
2. (unbounded array spec.) domain equation $arr_2 = location \rightarrow val$
3. (bounded, atomic array spec.) atomic domain arr_3 and axioms, such as $1 \leq i \leq 100 \supset retrieve(store(a, i, v)) = v$
4. (bounded array spec.) domain equation $arr_4 = lb \otimes ub \otimes arr_1$

Assume val is interpreted as val. Unbounded stacks can be implemented by unbounded array-integer pairs. For example, stk_1 can be interpreted as $arr_1 \otimes int$, where $is\text{-}stk(\langle a, i\rangle)$ iff $i \geq 0$, and $\langle a, i\rangle =_{stk} \langle a', i'\rangle$ iff $(i = i' \ \& \ (i > 0 \supset (1 \leq j \leq i)\ retrieve(a, j) = retrieve(a', j)))$. Similarly, it can also be interpreted as $arr_2 \otimes int$, where $is\text{-}stk(\langle a, i\rangle)$ iff $i \geq 0$, and $\langle a, i\rangle =_{stk} \langle a', i'\rangle$ iff $(i = i' \ \& \ (i > 0 \supset (1 \leq j \leq i)\ a(j) = a'(j)))$. This assumes $=_{stk}$ is inclusive.

Now, consider stk_2. Using the interpretation defined in this paper, stk_2 is interpreted as val^*. The specification stk_2 restricted the set of possible implementations. However, this can be slightly relaxed because if (1) $is\text{-}stk(\langle a, i\rangle)$ iff $i \geq 0$ and (2) $\langle a, i\rangle = \langle a', i'\rangle$ iff $(i = i' \ \& \ (i > 0 \supset (1 \leq j \leq i)\ a(j) = a'(j)))$, then val^* is isomorphic to the quotient of the is-stk subset of $arr_2 \otimes int$. Define the isomorphism $\theta: U_{val^*} \rightarrow U_{is\text{-}stk}/U_{=_{stk}}$ as $\theta(\langle\rangle) = \{\langle a, 0\rangle\}$ and $\theta(\langle v_1, \ldots, v_n\rangle) = \{\langle a, n\rangle \mid (1 \leq j \leq n)\ a(j) = v_j\}$. . Similarly, stk_2 can be implemented as $arr_2 \otimes int$. The domain stk_3 cannot be implemented as any array-integer pair because stk_3 allows infinite length stacks and array-integer pairs represent finite length objects.

The bounded stacks stk_4 and stk_5 can be represented as any of the arrays. For example, stk_5 can be implemented as $arr_1 \otimes int$ where $is\text{-}stk(\langle a, i\rangle)$ iff $0 \leq i \leq 100$. It can be implemented as $arr_3 \otimes int$ where $is\text{-}stk(\langle a, i\rangle)$ iff $i \geq 0$. And it can be implemented as $arr_4 \otimes int$ where $is\text{-}stk(\langle\langle l, u, a\rangle, i\rangle)$ iff $l = 0$, $u = 100$ and $i \geq 0$.

8.2. Interpretation Alternatives

The interpretation defined in this paper is a relatively simple extension of the interpretation defined in [wand 82a]. The designer is allowed considerable freedom in interpreting atomic domains. However, the interpretation of derived domains is defined in terms of constituent domain interpretations. This bottom-up method of domain implementation is described in detail above. This process ensures that the source domain equations will be satisfied in the implementation.

Some simple interpretation alternatives are discussed in this section.

8.2.1. Interpreting an Atomic Domain as a Function Space

An obvious extension of the interpretation defined in this report is to allow an atomic domain D to be represented by a function space of atomic domains in the target theory where the entire function space contains legal source representatives (i.e., is-D(d) = TRUE for all d) and each value in D has one representation in the target (i.e., $=_D$ is $=_{I(D)}$). In this simple extension, is-D and $=_D$ are inclusive predicates. Thus, a source structure can be derived from the target structure and the interpretation.

If the entire target function space does not represent source values, or if individual target function space values do not represent unique source values, then the predicates is-D and $=_D$ are not the trivial cases described above. The designer would have to prove that the predicates exist and are inclusive.

8.2.2. Top-Down Domain Interpretation

Initially, the designer may wish to ignore the composition of a derived domain and define its interpretation irrespective of the interpretation of the constituent domains. For example, for the compiler problem the designer may know that a source environment is represented by some target environments and initially ignore the fact that these environments are highly structured domains. However, the designer must eventually ensure that this interpretation is consistent with one developed in a bottom-up manner; the implementation of a derived domain must be consistent with the implementation of its constituent domains. Two examples are considered below.

First, take the case where a designer decides that the source domain of programming language environments, call it state, should be implemented by some target domain of machine language environments, call it mstate. In the source theory there is the domain equation state = memory ⊗ input ⊗ output. In the target theory there is the domain equation mstate = stack ⊗ memory ⊗ input ⊗ output. The bottom-up interpretation process yields the interpretation of state as (memory ⊗ input ⊗ output). The process is easily relaxed where state can be interpreted as mstate. This requires that two mstate values be state-equivalent if and only if their memory, input, and output projections are equal. The reason is $U_{memory \otimes input \otimes output}$ is isomorphic to $U_{mstate} / U_{\approx_{stack}}$; the bottom-up interpretation is isomorphic to the top-down interpretation. The top-down interpretation is preferable because the target operators are specified in terms of mstate.

Consider another compiler application example. Say in the source theory there are domains for statement continuations, cont, and expression continuations, econt. The syntactic structure for the programming language specified by the target theory is simpler than that specified by the source theory. In the target theory there are only machine instruction continuations, mcont. Using the specifications of state and mstate in the previous example, the continuation domains are defined by:

1. cont = state → (state ⊕ error)
2. econt = (value → cont)
3. mcont = mstate → (mstate ⊕ error)

Assume the designer decides that the source continuations cont and econt are implemented by some particular partitioned subsets of mcont. Also assume that there is no representation change for value; value is interpreted as value.

The domain equations for cont and mcont are similar. There is no problem with cont as mcont because the bottom-up interpretation is I(state → (state ⊕ error)) = (mstate → (mstate ⊕ error)). Values in mcont are restricted to those that accept or return source representative values in mstate because is-cont(z) iff is-(state → (state + error))(z).

The representation of econt in mcont is not as straightforward because the domain equations differ in syntactic structure. An expression value is an intermediate result that is passed to the rest of the program. At the target level, an intermediate result is an environment, mstate, which is passed to the rest of the program. A bottom-up interpretation of econt is (value → mstate → (mstate ⊕ error)). This is isomorphic to (value ⊗ mstate → (mstate ⊕ error)). Assuming the stack component of mstate is isomorphic to value*, then the interpretation of econt is isomorphic to ((value ⊗ (value* ⊗ memory ⊗ input ⊗ output)) → (mstate ⊕ error)). Call this $econt^I$. The domain $econt^I$ is isomorphic to a subdomain of mcont. This is important because the source operator f: econt → D can be interpreted as a term I(f): mcont → I(D), where I((f)(x)) is (I(f))(I(x)) and I(x) is implicitly coerced to type mcont via retractions between $econt^I$ and mcont.

In the interpretation of cont above, the domain restriction was stated explicitly in $=_{cont}$. The domain restriction was derived inductively from constituent domains for the interpretation of cont. The domain restriction for the interpretation of econt was implicit because the interpretation is a subdomain of an existing target domain.

8.3. Direct/State Tiny - State Interpretations

In this section and the following section, implementations of the programming language Tiny, as defined in [gordon 79a], are discussed. Tiny has identifiers, expressions, commands, and programs as programming language constructs. In this section, the semantics of constructs are defined in terms of state changes. A direct semantic description means the description does not have continuations. This is addressed in the next section.

The execution of each command of Tiny results in a state change. The state has three components:

1. *memory*: this is a correspondence between identifiers and values. In the memory each identifier is either bound to some value or to **unbound**.
2. *input*: this consists of a (possibly empty) sequence of values which can be read using the expression **read** and is supplied by the programmer before the program is executed.
3. *output*: this is an initially empty sequence of values which records the results of the command **output**.

The meaning of an expression is a value-state pair, where a value is either a boolean or a number. Because expressions may contain identifiers, the value depends on the state. The meaning of a program, given some input, is some output or an error.

Refer to the direct/state semantic description of Tiny as *DS-Tiny*. DS-Tiny is formally specified as a source theory in Appendix C. The direct/state semantic description of the target theory is also specified. The interpretation is defined and part of the correctness proof is illustrated.

The target language for DS-Tiny has instructions and sequences of instructions (code) as programming language constructs. The syntactic hierarchy of the defined language is simpler than that of DS-Tiny. Refer to the target language as *DS-Tinytarget*. The "execution" of an instruction or code results in a change of the target (or machine) state, call it mstate. The target state is almost the same as the source state. It has as an additional component a stack. The stack is used in evaluating expressions.

In both DS-Tiny and DS-Tinytarget, if any of the constructs produces abnormal results, the error result must be passed to the program following it. This is what happens in a direct semantic description. The extra checking involved makes for a more complicated specification and may be unnatural because intuitively, when an error occurs the computation cannot be stopped, but must be continued. The continuation semantics of Tiny in the next section results in a more elegant and "natural" specification.

In both source and target specifications, the theory of domains, described in Chapter 6, is assumed and not written as part of the specification. This includes all domain operators, axioms, equality symbols, and logical symbols. However, the domain constructor * was not specified previously. Operators and axioms for it are specified in each theory. Also, instead of using operators isl and isr on sum domains, we use, for example, isnum: (num $\oplus$ bool) $\rightarrow$ bool for $isl_{num\ bool}$, etc.

The interpretation from the language of the theory for DS-Tiny to the language of the theory for DS-Tinytarget is also specified in Appendix C. The defined language (abstract syntax) of DS-Tiny is interpreted as the defined language of DS-Tinytarget, and the defining language (semantic domains and operators) is interpreted as the defining language of DS-Tinytarget. For example, the operator symbol + in the source defined language has signature (exp $\otimes$ exp $\rightarrow$ exp). The interpretation, denoted $\mathcal{I}$, of + is the term ($\lambda E_1\ E_2\ .\ E_1 \bullet E_2 \bullet$ [**add**]) with signature (ecode $\otimes$ ecode $\rightarrow$ ecode) where $\mathcal{I}$(exp) = ecode. Thus, the source term $+(I_1, I_2)$ is interpreted as ($\mathcal{I}(I_1) \bullet \mathcal{I}(I_2) \bullet$ [**add**]). An addition expression with two constituent expressions is implemented by implementing each of the constituents and then executing the instruction [**add**].

Another example, is the interpretation of state, a domain in the source defining language. The domain state is interpreted as mstate. This is described in detail above in Section 8.2.2. The semantic operator for commands, ***C***: com → (state → (state ⊕ {error})), is interpreted as the term λCs.***MC***(C)(s) with signature (code → mstate → (mstate ⊕ {error})). The interpretation of the semantic operator for expressions, ***E***: exp → (state → ((value ⊗ state) ⊕ {error})), is more difficult. It is interpreted as (λEs.***H***(E)(s)) with signature (ecode → mstate → ((value ⊗ mstate) ⊕ {error})), where ***H*** is a new operator symbol and ***H*** is defined in terms of ***ME***.

Part of the correctness proof is also in Appendix C. Ignoring the preambles (they are trivially satisfied), the source axioms are translated using the interpretation, and then the translated axioms are deduced in the target theory. The translation essentially involves using the definition of ***I*** and β-conversion. The deduction of the translated axioms in the target theory primarily uses the semantic equations of the target theory as rewrite rules. Most of this is routine and could be mechanized. The creative part of the proof arises when the target theory does some checking that is not evident in the translated axiom. For example, in the target theory, various instructions (e.g., [**not**], [**eq**]) operate on the stack. Prior to execution, the stack is checked to see if it meets certain conditions (e.g., the top of the stack is checked for a boolean value prior to executing [**not**]). The axioms at the source level do not refer to any expression stack. Therefore, it must be proved that those required stack conditions are always true in the implementation. These conditions are proved by (explicit) structural induction in a set of lemmas, also in Appendix C.

Axioms (E1a) to (E5) are discussed in the correctness proof in the Appendix. All the axioms are presented in the proof for the implementation of the continuation/state description of Tiny. This is reviewed in the next section.

8.4. Continuation/State Tiny - Continuation Interpretations

In the previous section an implementation of Tiny was described where the semantic description was written in a direct style. In this section the semantic description of the same programming language is written in a continuation style (sometimes referred to as standard semantics). With continuations, denotations do not transform states directly, but rather, transform states indirectly though continuations. A continuation is a domain that models control. They were initially developed to model unrestricted branches (gotos), but since then, have been useful for modelling other nonstandard evaluation orderings. The simplification strategies for function notation are sometimes mistakenly taken for the program sequencing strategy (the operational evaluation). This is fairly innocuous when the order of evaluation is not important. But, some programming languages provide the programmer with the ability to change the order of evaluation. For Tiny, continuations allow immediate program exits when error conditions are raised.

It should be noted that in [reynolds 74] it was shown that direct semantics are included in continuation semantics. So, any direct semantic specification can be rewritten with continuations. Thus, it is reasonable that we require that both the source and target theories be specified with the same semantic style. However, some proofs of congruence between direct and continuation semantics are quite difficult.

Refer to the continuation/state description of Tiny as CS-Tiny. This, along with the target theory specification, the implementation, and the proof, are in Appendix D. In CS-Tiny, there are two kinds of continuation domains, one for commands, denoted cont, and one for expressions, denoted econt. As explained in [gordon 79a], a continuation is a function from whatever the "rest of the program" expects to be passed as an intermediate result to the "final answer" of the program. The continuation represents "the remainder of the program." A command expects a state as an intermediate result and the final answer is either a state or an error message. Thus, cont is defined as (state → (state ⊕ error)). On the other hand, an expression expects a value as an intermediate result and this is embedded in a command. Hence, econt is defined as (value → cont).

The semantic operator for commands is ***C***: com → cont → cont. The meaning of a command is a function of a continuation and a state which yields the final answer of the program (a state or an error message). The semantic operator for expressions is ***E***: exp → econt → cont. The meaning of an expression is a function of an expression continuation and a state which yields the final answer to the program.

Refer to the target theory for CS-Tiny as CS-Tinytarget. CS-Tinytarget is similar to DS-Tinytarget, except that continuations are used. A continuation in the target theory, denoted mcont, is a function from the machine state to either a machine state or an error message. So, mcont = (mstate → mans) = ((stack ⊗ state) → ((stack ⊗ state) ⊕ error)). The meaning of an instruction or a sequence of instructions is a function of a machine continuation and machine state which yields a machine state or an error.

The defined language and the interpretation of the defined language for CS-Tiny are identical with that for DS-Tiny. The interpretation of the defining language for CS-Tiny includes interpreting state as mstate (same as for DS-Tiny), cont as mcont, and econt as a subdomain of mcont. This is described in detail in Section 8.2.2. Notice in particular the interpretation of k: econt. The domain econt is isomorphic to (value → state → (state ⊕ error)). Its interpretation, (value → mstate → (mstate ⊕ error)), is isomorphic to a subdomain of mcont. The variable k is interpreted as the term (λv(stk, m, i, o).z((v • stk, m, i, o))): (value → mstate → (mstate ⊕ error)), where z has signature mcont. The interpretation of k can also be uncurried so that it is (λ(v • stk,m, i, o). z((v • stk, m, i, o))). If v • stk is replaced with some other variable, say stk, then the whole term can be rewritten as z.

The correctness proof proceeds as in the proof for DS-Tiny. It involves deducing interpreted source axioms in the target theory.

There is also another continuation semantic definition of Tiny where econt is recursively defined as econt = cont ⊕ (value → econt). In this specification the implicit notion of an expression stack is seen more clearly. Refer to this description of Tiny as CS-Tiny2. Its specification is in Appendix E. Using the interpretation described in this paper, econt cannot be mapped into a subdomain of mcont. However, if econt is unfolded where $econt_0$ = cont and $econt_{n+1}$ = value → $econt_n$, then we can define an interpretation as above. The terms have ellipses in them and an appropriate interpretation must be found. For example the axiom:

E[[**read**]](k) $=_{econt}$
λv_1 ... v_n (m, i, o). null(i) → **empty-input**,
k(***hd***(i))(v_1) ... (v_n) ((m, ***tl***(i), o))

would be interpreted as;

ME([**read**])(z) $=_{mcont}$
λ(<v_1 ... v_n>, m, i, o). null(i) → **empty-input**,
z((<***hd***(i)> • <v_1 ... v_n>, m, **tl**(i), o))

8.5. Continuation/State Small - Declaration and Procedure Interpretations

In this section an implementation of the programming language Small, as defined in [gordon 79a], is discussed. In addition to identifiers, expressions, commands, and programs, Small has declarations. The declarations allow programmer defined constants, variables, and, procedures. Small, as defined in [gordon 79a], also has functions. We eliminated this from the language because it is similar to procedures. The semantic description in [gordon 79a] is written in a continuation/store style. The semantic description in this section is written in a continuation/state style and is a natural extension of the CS-Tiny specification. It is referred to as CS-Small. The specification and implementation of CS-Small are in Appendix F.

In CS-Small there are three types of continuations; there are continuations for commands (cont), for expressions (econt), and for declarations (dcont). A state consists of:

1. an *environment*: this binds identifiers to denotable values or to **unbound**. The denotable values are locations, boolean or basic values, or procedure values.
2. a *store*: this binds storable values to locations. The storable values are the input file and boolean or basic values.
3. an *answer*: this is a sequence of boolean or basic values followed by either **error** or **stop**. This denotes the total output of a program.

The domain dv is the set of denotable values. The continuations are defined as follows:

1. cont = state → state
2. econt = dv → cont
3. dcont = env → cont

The meaning of a command or an expression is similar to that in CS-Tiny. The meaning of a declaration is a function of a declaration continuation and an environment and yields a state-to-state transformation. The domain for procedure values, proc, is defined as (cont → (dv → cont)); a procedure value, given a continuation (the "rest of the program" following the procedure call) and a denotable value (the actual parameter to the procedure), returns a continuation (the "rest of the program" with a modified state).

The syntactic hierarchy of the defined language of the target for CS-Small, referred to as CS-SmallTarget, is simpler than that of CS-Small. CS-SmallTarget contains instructions and sequences of instructions as programming language constructs. Consequently, there is only one kind of continuation domain, mcont. As in CS-TinyTarget, mcont is a function space from mstate to mstate. The mstate for CS-SmallTarget is a bit more complicated than that for CS-TinyTarget. It has five components:

1. an *environment*: this is a stack of local environments (activation records or association lists). Local environments are distinguished by **begin/end** instructions, the environment is altered in **bind** and **mkproc** instructions, and the environment is accessed in the **load** instruction.
2. a *store*: this is essentially the same as the store for CSSmall.
3. an *answer*: this is essentially the same as the answer for CSSmall.
4. a *stack*: this is a stack of denotable values for evaluating expressions.
5. a *dump*: this is a stack of environments. Environments are pushed when a procedure is activated and the dump is popped before returning from a procedure activation.

The interpretation and part of the correctness proof are also in Appendix F. It is similar to the interpretation of CS-Tiny in that different types of continuation domains at the source level are interpreted as some subset of a continuation domain at the target level. Also, the source state domain is interpreted as the target state domain. The concept is the same as that for CS-Tiny with the exception that state =

(env ⊗ store ⊗ ans), mstate = (menv ⊗ store ⊗ ans ⊗ stack ⊗ dump), and the interpretation of env is not menv, but, rather, the interpretation of env is isomorphic to a subdomain of menv. Specifically, the interpretation of env is (id → (mdv ⊕ {**unbound**})). This is not isomorphic to (id ⊗ mdv)*. However, we would like to implement the function space as the nonfunctional domain, the conversion sometimes referred to as defunctionalization. It is easy to see how any function in the function space can be represented in the nonfunctional space. For example, the undefined function f (for all i in id, f(i) = **unbound**) is represented by the empty list, <>. The function f, defined at I_1 and I_2 such that $f(I_1) = e_1$ and $f(I_2) = e_2$ is represented by $<<I_1, e_1>, <I_2, e_2>>$. The nonfunctional domain is larger than the functional one. An equivalence relation is defined on it in order to map it back to the functional domain. Intuitively, only one mdv element must be paired with each id element. Hence, $<<I_1, e_1>, <I_1, e_2>>$ is isomorphic to $<I_1, e_2>$, and $<I_1, e_2>$ is a representation for the function f, such that $f(I_1) = e_2$. The domain (id ⊗ mdv)* is isomorphic to alist, and alist is isomorphic to alist ⊗ {<>}. The domain (alist ⊗ {<>}) is isomorphic to a subdomain of menv. Intuitively, at the source level, the "current" environment is one list. At the target level, the "current" environment is a stack of lists. The concatenation of all these lists into one list does not affect the semantics. In particular, when evaluating a **load** instruction the stack of lists is accessed in the same order as would a list constructed by concatenating the list; the local variable is closer to the top of the stack or the beginning of the list.

The interpretation of the defined language is similar to that of CS-Tiny. However, CS-Small has additional constructs, such as declarations and procedure calls. For example, the interpretation of the procedure declaration **proc**(I, I_1, C) is the code [**mkproc**, [**bind** I_1] • *I*(C) • [**ret**]] • [**bind** I]. The interpretation of the procedure call $E(E_1)$ is *I*(E) • *I*(E_1) • [**pcall**].

The interpretation of the semantic operators are also similar to that of CS-Tiny. For operators with econt or dcont in the signature, the interpretations are terms in which the mdv or menv arguments are "absorbed" into mstate by the usual uncurrying method.

Note that abbreviations are used in the axioms. These are defined following the axioms. In particular, deref takes an argument of type econt and returns an argument of type econt. If the denotable value passed to the econt object is not a location, then the econt object is returned. If the denotable value passed to the econt object is a location and that location in the store is not **unused**, then the value in the store is made the argument to the econt object; the denotable value is dereferenced. The abbreviations are interpreted. The interpretation for deref is given the name deref^T.

The abbreviation deref is used for the ("right-hand-side") meaning of a source expression and is given by the operator ***R***. The operator ***R*** is defined in terms of ***E*** and deref. Thus, the right-hand-side meaning of an expression is a function that takes either a boolean or basic value. The ("left-hand-side") meaning of an expression, given by ***E***, is a function that takes boolean or basic values, in addition to locations. This is why the interpretation of ***R*** is defined in terms of ***ME*** and deref^T, while the interpretation of ***E*** is defined in terms of ***ME***.

Chapter 9

Comparison of Compiler Design Verification Methods

The compiler correctness problem has been considered an important application of formal verification from the beginning of verification research. This chapter briefly reviews the progress by characterizing previous research in terms of the semantic definition method and proof organization used. This may oversimplify previous work, but, it is beyond the scope of this dissertation to give a detailed comparative analysis. A comparative analysis of different verification methods and systems of the last twenty years would in itself be an interesting and useful research topic. The purpose of this discussion is to gain some perspective on the topic and determine how the work discussed in this dissertation relates to other research.

The verification methods can be differentiated by the specification languages (or logics) used and how two specifications are related. For the compiler problem, each specification contains the syntax and semantics of a programming language. The choice of specification language effects the types of relationships that can be defined and the correctness proof organization. Hence, it effects whether the method is conceptually clear, whether it can be automated, and whether it can be used for real, large applications.

In this chapter, compiler design verification methods are distinguished by whether the specification is based on (1) denotational, algebraic, or axiomatic semantics or (2) operational semantics. Assume the abstract syntax of the programming language is specified. The basic idea is presented in Figure 9-1.

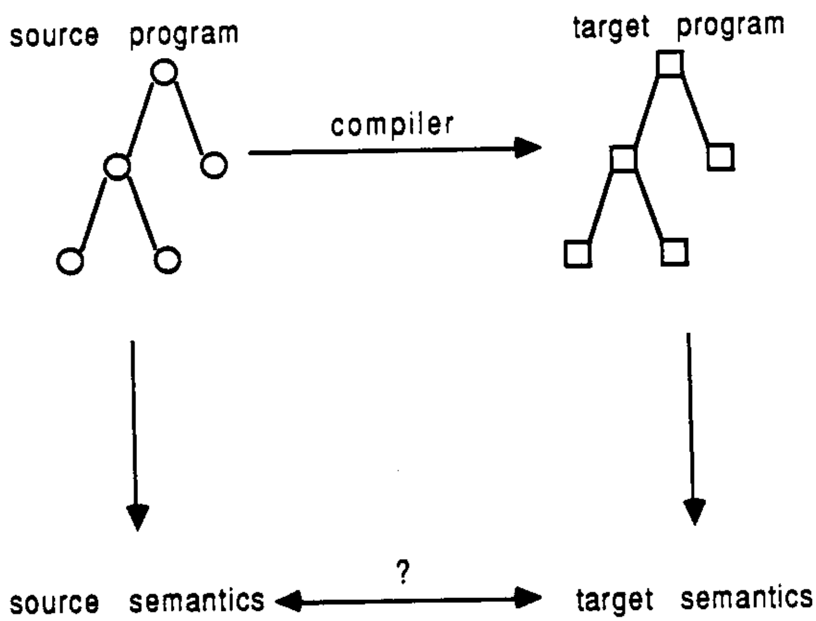

Figure 9-1: Compiler Design Problem

Assuming a non-optimizing, syntax-directed compiler, the translated source program results from the translation of each construct's constituents. This is indicated by the tree structures in the figure. The semantics of the source program and the translated source program must be related. For the compiler design to be correct, the diagram must commute for *any* source program. Hence, an induction argument must be made over all source programs.

If the semantics is written in a denotational or algebraic language then a structural induction argument on the source syntax can be made to determine whether the source and target are related. This is illustrated in Figure 9-2.

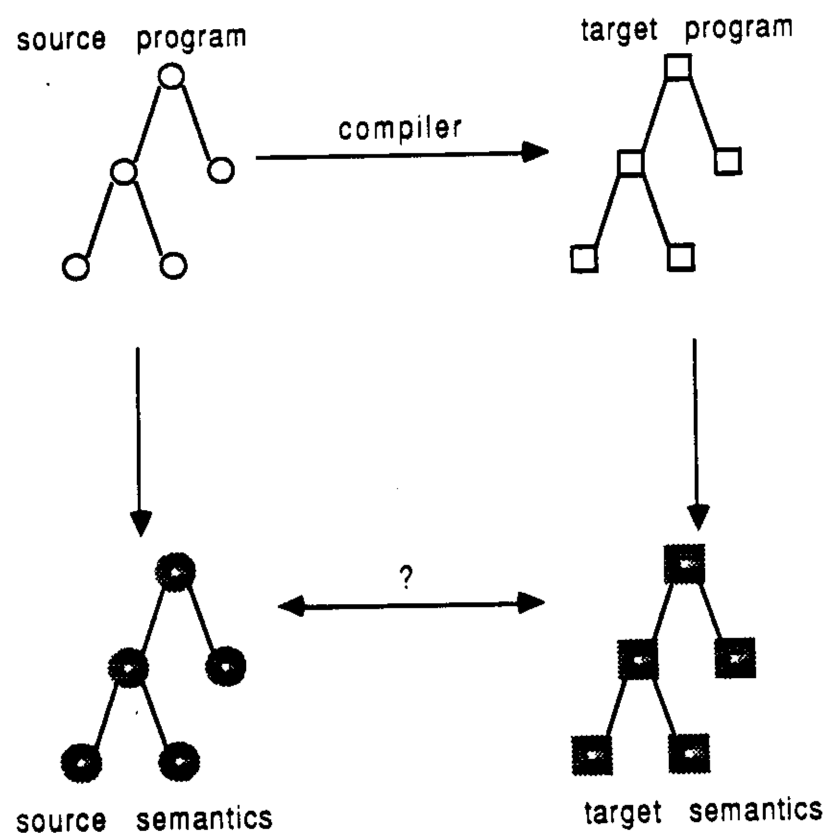

Figure 9-2: Compiler Design Problem Based on Denotational Semantics

Figure 9-3 crudely illustrates the problem for operational semantics. An abstract machine, or interpreter, is defined for each language. It must be shown that any compiled program when executed, has the same effect as the source program would have if it could have been directly executed.

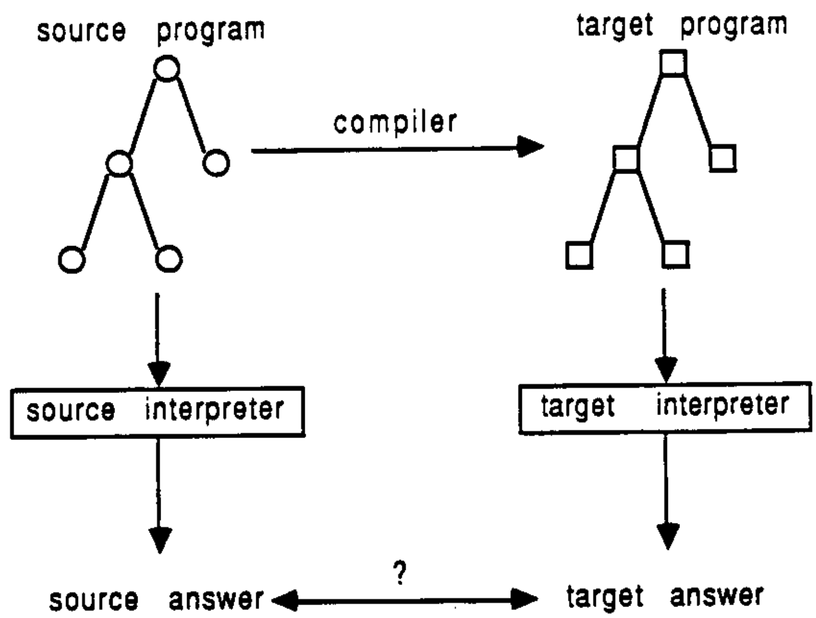

Figure 9-3: Compiler Design Problem Based on Operational Semantics

9.1. Using Denotational, Algebraic, or Axiomatic Semantics

Methods using denotational or algebraic semantics have been based primarily on the commutative diagram in Figure 9-4. Reports on such research include [milner 72], [morris 73], [chirica 76], [thatcher 79], [mosses 80], [cohn 81], [polak 80], [dybjer 83], [milne 83], [orejas 84], [royer 86], and [despeyroux 86]. Several references propose that the bottom arrow of the diagram be directed from left to right. This conflicts with our premise that a source object (meaning) can have two or more equivalent representations. Reference [orejas 84] also agrees with this requirement.

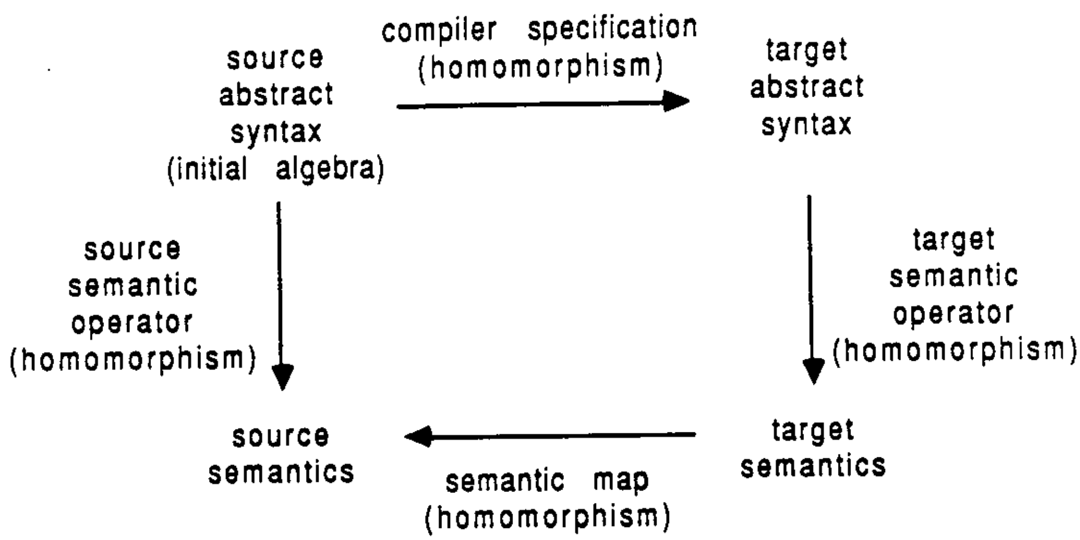

Figure 9-4: Algebraic Technique

To prove the commutativity of the diagram in Figure 9-4 it is sufficient to prove that the semantic map is a homomorphism because the compiler specification, the source semantics, and the target semantics are all defined as homomorphisms. The commutativity results from the initiality of the algebra specifying the source syntax. The overall correctness proof is based on structural induction on the source syntax. The structural induction comes from the initial algebra property. Other types of induction may be used to prove some subgoals.

The structural induction argument is used explicitly when one proves the semantic map is a homomorphism. There is a commutative diagram for every source syntactic domain, and hence, one for every source construct. Complex syntactic types are syntactic types that have other syntactic types as proper constituents. The semantic maps for complex syntactic types are proved assuming the diagrams for constituent syntactic types. The proof consists of an interleaving of term simplification using *both* source and target properties, and the induction hypotheses.

The semantic algebras can be mapped to other algebras, referred to here as models or structures and illustrated in Figure 9-5.

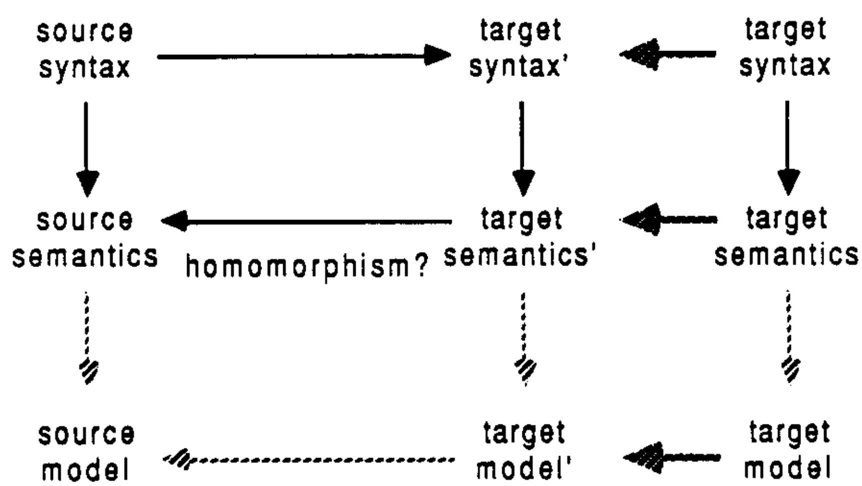

Figure 9-5: Algebraic Technique

A denotational semantics might be mapped to cpo's and continuous functions. The models are not relevant to particular correctness proofs, but, should be defined in general. The illustration also brings to light both the subsets and quotients inherent in an implementation.

Related to the algebraic approach is the approach presented in this dissertation. It is a different paradigm for the compiler problem, where a design or implementation is specified as an interpretation. The overall correctness proof is again based on structural induction on the source language, but it is not justified in terms of initial algebra arguments. The proof itself is different in that it consists of a translation and then a simplification using target properties. Instead of proving that a semantic map is a homomorphism, translated formulas are deduced in the target theory. This is depicted in Figure 9-6.

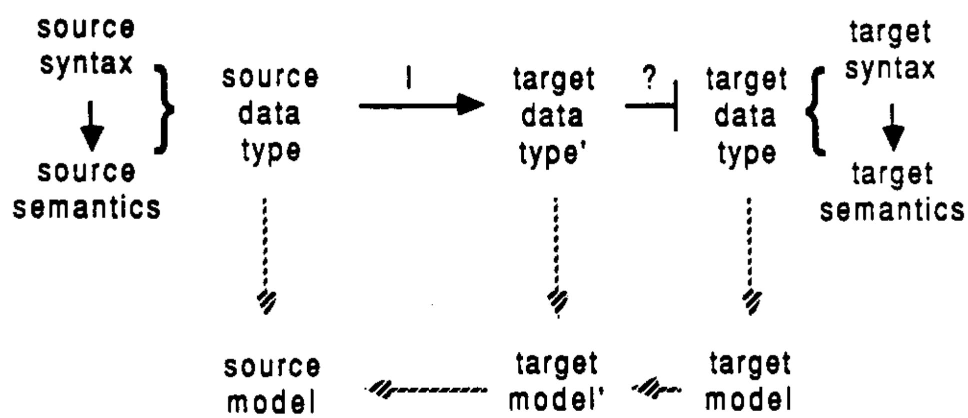

Figure 9-6: Interpretation Technique

Again, the models are not relevant in individual correctness proofs, but are used in this dissertation to show the correctness criteria are complete.

So, how does the interpretation technique compare with the algebraic technique? It has been noted in [polak 80] that the concept of homomorphism is hard to understand and it is difficult to formalize the concept for current verification systems. The particular proofs involve an interleaving of induction steps, which is difficult to automate. The concepts of logical theories, mappings, and deduction discussed in this dissertation are well-known and in some sense, very intuitive. Many of the proofs involve fairly easy, but tedious, term rewriting.

Everything considered, the comparison is subjective, especially when one tries to determine what method is a better way to formally specify and organize mental thought processes. Informally, a representation is constructed via a mental comparison of the intended behavior of a concept and the actual behavior of the implementing environment. An implementation is how the representation is constructed. An implementation is correct if the representation constructed preserves the concept behavior. Therefore, to formalize the verification process one needs formal descriptions of:

1. the abstract concept
2. the implementing environment
3. the implementation
4. the criteria a correct implementation must satisfy

The interpretation method is proposed because these requirements are naturally expressed. The abstract concept and implementing environment are specified as abstract data types (theories) and an implementation is perceived as a mapping from one data type to another. What is particularly important for the compiler problem is that there is a *clear* distinction among programming language syntax, programming language semantics, theory syntax, theory models, and implementations. Wand, in [wand 82a], notes that the distinction between specifications and modelling is particularly difficult in an algebraic framework.

A few other references should also be noted in this section. The books [milne 76] and [stoy 77] are standard works on denotational semantics and discuss the compiler correctness problem. The correctness proofs are based on explicit structural induction on the source language and the semantics are related by inclusive predicates. Their predicates are more general than the ones allowed in this research, but the correctness proofs are much harder. This was discussed earlier. As in the algebraic method, the distinction between specifications and modeling is not sharply defined. However, this research would have been impossible without their work and the algebraic work.

Lastly, nothing has been mentioned about axiomatic semantics. Little has been done with first-order programming logics to solve the compiler design correctness problem. It is mentioned in this section because [lynn 78] discusses a compiler proof using Hoare logic. Lynn's proof of a LISP compiler is a formal, mechanized version of a proof done by London in [london 71]. London's proof is based on operational semantics and is mentioned in the next section. The partial correctness formula of the form P{A}Q, where P and Q are predicates and A is a program, is true if and only if for all states s and s', P is true given s and <s, s'> is in the relation assigned to A, implies Q is true in s'. The state <s, s'> is in the relation if and only if A executed in s can terminate in s'. However, states are not represented in the partial correctness formulas. This leads to rather unnatural semantic definitions. In particular, the Hoare logic semantics of function routines are hard to understand because other indirect notation must be introduced to convey the properties of scope and parameter passing. New variables are introduced to denote a value before execution versus after execution.

Lynn's approach is related to the interpretation method in that the partial correctness formulas for the source language are translated into the target language, and then the translated source axioms are proved true. However, the LISP example chosen is very simple. The translation involves changing variable names into locations and source language constants into their target representation. The structural induction argument is also very simple because the source language is a subset of pure LISP and there are no assignments or global variables.

It should be noted that first-order logic verification systems were used to mechanically check the compiler *implementation* proofs in [polak 80] and in [lynn 78]. However, [polak 80] initially uses denotational semantics and defines the problem within the algebraic framework discussed above. Also, [chirica 86] uses an algebraic framework to present a method for proving the correctness of parse-driven implementations. It is algebraic in nature, but uses attribute grammars as a means of obtaining an algebraic specification. Sequences of compiler translation routines are proved partially correct via the standard inductive assertion method. These references are noted, but, not reviewed because it is the compiler design problem that is the primary issue in this dissertation.

9.2. Using Operational Semantics

With operational semantics, the meaning of a program is given by a sequence of computation states that results from executing the program on an "abstract machine". Hence, the meaning of a construct may depend on more or something else than the meaning of its constituent constructs. For example, in operational semantics the meaning of a procedure may be represented by a structured object, sometimes called a closure, which contains, among other things, the text of the procedure body. In contrast to denotational semantics, textual information is operated on and passed to various functions.

This is made clear in [stoy 77] where the following simple example is presented:

1. Domains:
   Bas

   B

   Exp

   Id

   U= Id → Exp
2. Operations and variables:
   B: Bas → B

   E: Exp → U → B

   ρ: U

   b: Bas and b: Exp

   I: Id and I: Exp

   λ: Id ⊗ Exp ⊗ Exp → Exp
3. Axioms:
   $\boldsymbol{E}\,(b)(\rho) = \boldsymbol{B}\,(b)$

   $\boldsymbol{E}\,(I)(\rho) = \boldsymbol{E}\,(\rho(I))(\rho)$

   $\boldsymbol{E}\,((\lambda I.\ E_0)E_1)\ (\rho) = \boldsymbol{E}\,(E_0)\ (\rho[E_1/I])$

At first this appears to be a denotational description. However, upon close examination of the second axiom one notices that ***E*** is not a homomorphism; the meaning of the identifier I does not just depend on constituents because ρ(I) is not a subcomponent of I. The meaning of I is defined in terms of the meaning of ρ(I), which can denote more text. A typical specification of ***E*** as a homomorphism is $\boldsymbol{E}(I)(\rho) = \rho(I)$ where the state ρ returns a semantic value when given an identifier.

So, two questions arise from this example. Is it an operational definition? If so, how does one use it in a correctness proof? The word "operational" is ambiguous. A denotational definition can be considered operational when rewriting and β-conversion rules are used. Sequences of computation states could correspond to the sequence of rewriting and simplification steps. However, we prefer to draw the line at whether or not the meaning of a language construct depends solely on the meaning of its constituents. Hence, the example above is an operational definition and was in fact, derived from an abstract machine definition in [stoy 77].

It is important to show that an operational semantics definition is well-defined. Unlike denotation semantics, it is not trivial to justify a definition. If the example above was changed so that ***E*** is a homomorphism and the atomic components are well-defined, all components are well-defined. Because ***E*** is not a homomorphism in the example above, structural induction cannot be used to show that ***E*** is well defined. If there is no choice of evaluation (simplification), then one shows the definition of ***E*** is not circular. If there is a choice of evaluation, then one must show that all evaluations of a term reduce to equivalent terms; it is not well-defined if two evaluations of the same term return inequivalent results. Furthermore, a correctness proof that uses operational definitions must be based on induction over the computation steps, rather than on induction over the source syntax.

As mentioned above, the example was derived from an abstract machine specified in [stoy 77]. The machine is defined as a process that modifies a state at each step until a terminal state is reached. If the domain of states is S, then a function step with signature S → S and a predicate term with signature S → bool are defined where step specifies the state transition and term specifies terminal states. A function machine with signature ((S → S) ⊗ (S → bool)) → (S → S) is defined such that machine(step, term) = Fix(λfs. term(s) → s, f(step(s))). To define any particular machine, definitions of step and term are given.

This is similar to the Information Structure Model (ISM) in [wegner 70] which abstracts other operational semantic definition methods such as the contour model [johnston 71] or the Vienna Definition Language (VDL) [lucas 70]. An ISM is defined as a triple M=(I, I_0, F) where I is the set of all possible computation states, I_0 is the set of initial states (a subset of I), and F is a transition function on I to subsets of I. Hence, the only significant difference between ISM'S and Stoy's definition is that an ISM allows nondeterminism. However, Stoy's definition can be modified to allow nondeterminism via power domains (similar to a powersets). Alternatively, an ISM can be defined deterministically. In the ISM M, a sequence C= <S_o, S_1 ... , S_n> is called a computation if and only if:

1. for all S_i in C, S_i is also in I
2. if C is not the empty sequence, then S_0 is in I_0.
3. for all S_i in C such that $i \neq 0$, S_i is in $F(S_{i-1})$
4. C is not a proper initial sequence of any other sequence satisfying (1), (2), and (3) above.

Typically, a computation state includes such information structures as stacks, counters, pointers, registers, etc., and the transition function is defined as a computer program.

Using this paradigm, the compiler design correctness proof is an equivalence proof of source and target interpreters. This is sometimes referred to as the twin machine concept [lucas 68, mcgowan 72, wegner 72]. Let M be a deterministic ISM. Then $M(S_0)$ is either (1) undefined or (2) some projection of S_n when S_n is the final state in a computational sequence <S_0, S_1, ..., S_n>. Two interpreters M and M' are equivalent if the corresponding partial functions are equivalent; M and M' are equivalent if for all initial states S_0, (1) the M computation halts on S_0 if the M' computation halts on S_0, and (2) if the M computation halts, then $M(S_0) = M'(S_0)$. Of course, this assumes the state components for the two machines are identical which is unrealistic. Thus, a map or relation between machine states is required. More important, the general problem of proving two interpreters equivalent is undecidable. However, in practice, the problem is tractable because one does not deal with arbitrary ISM's, but one ISM is intentionally constructed to be equivalent to the other.

As described in [mcgowan 72], the proof technique, based on observation and confirmed by experience, is that if M′ is constructed with the intention that it be equivalent to M, then given input S_0, it is likely that some of the intermediate computation steps of M are related to some of the intermediate computation steps of M′. In practice, the proof becomes tractable by constructing mappings of the two computations which formally express intermediate relationships. This is illustrated in Figure 9-7

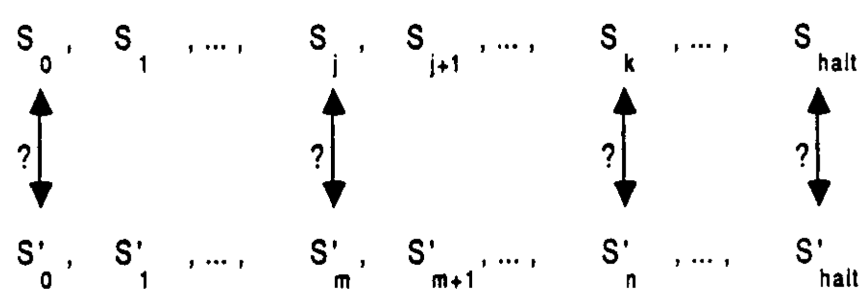

Figure 9-7: Twin Machine Technique

A more realistic illustration of what goes on in the proof is given in Figure 9-8.

There are two types of mapping. One type relates variables (or data structures) in one interpreter to variables in the other. The other type of mapping identifies and

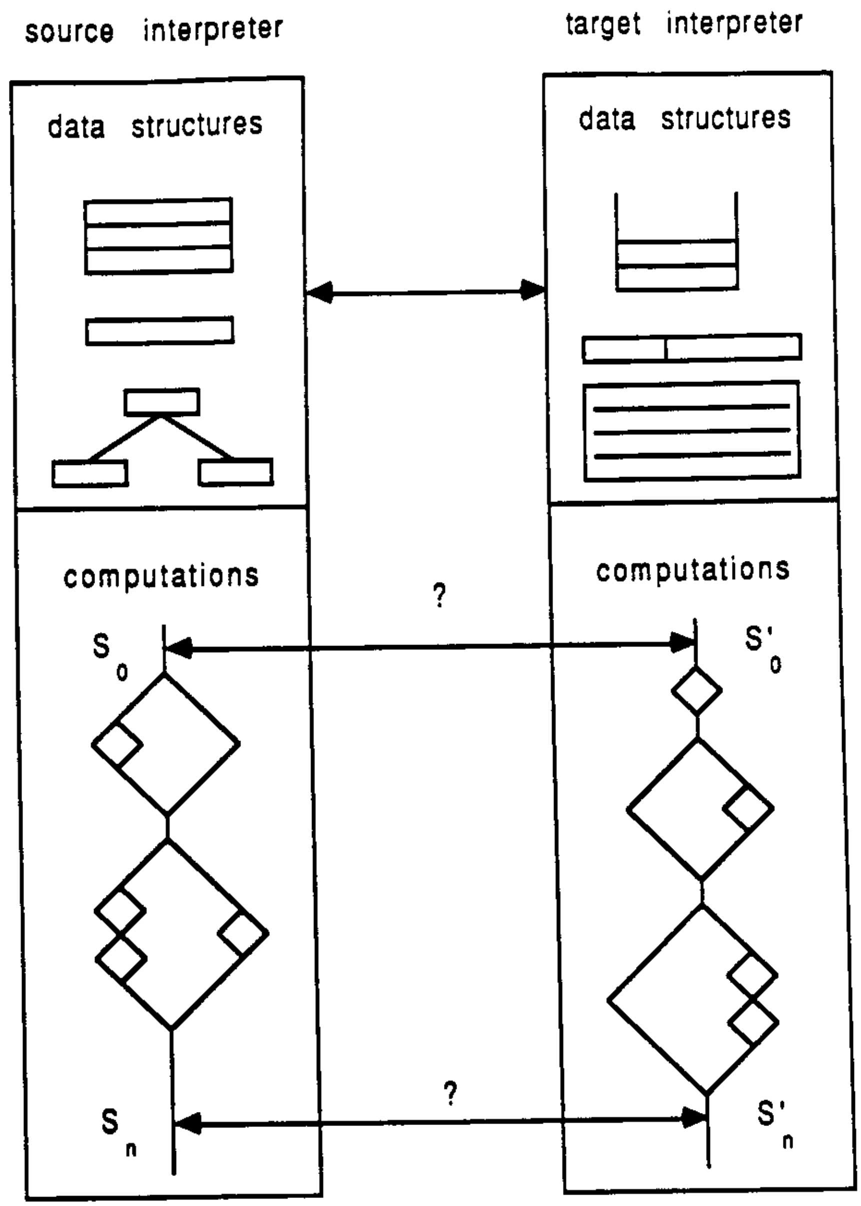

Figure 9-8: Proof Using Operational Semantics

relates intermediate computation steps. What should be apparent from the illustration, is that

1. The overall induction argument proceeds over computation steps
2. The semantics of individual programming language constructs must be abstracted from a large, complex algorithm

The major practical ramification of this is that if either specification (interpreter) is modified, it is difficult to determine what parts of a correctness proof must be redone. It requires a difficult analysis of all computation paths. Furthermore, it is difficult to decompose the verification process into small tasks that can be done independently and in parallel. Also, with operational semantics there is a tendency towards overspecification. Variables used to define an algorithm in one interpreter may have no counterparts in the other interpreter. The specifications tend to be very large. On a more subjective level, it has been argued that the proof process does not mirror the informal verification process which is based more on syntax-directed reasoning. Some of the discussion above can be found in [levy 84], [damm 85a], and [damm 85b].

On the other hand, there are advantages to using this methodology. Primary among them is that prototypes of verification systems based on first-order programming logics can be found, e.g., [stanford 79], [good 75], [marcus 84a]. Any attempt at a large application is almost impossible without some computer assistance. Any good verification system requires many person-years of development. The effort involved may be comparable to perhaps the development of an operating system or a compiler. This is not presented to give the impression that this type of verification is a solved problem. We are speaking about prototypes and ongoing research. A second advantage that has been proposed is that an operational definition is easier to write and anyone familiar with programming languages can read a definition. A third advantage is that it might be used successfully to verify implementations where the formal specifications and implementations are constructed independently, or where verification is done after the implementation. The other methods tend to require that the verification process be integrated into the design process. The last item to be mentioned is again rather subjective. The choice of semantic definition method may depend on what the source and target programming languages are. The source language could be rather low-level, e.g., assembly language, and thus, its meaning more intuitively corresponds to an abstract machine. However, higher-level languages are suppose to be "machine independent" and operational semantics tend to impose machine dependent properties. An interesting and influential discussion of this appears in [reynolds 72].

The general operational technique is discussed above, and now, some specific cases are briefly mentioned. Some of the earliest work on compiler correctness can in found in [mccarthy 67] and [painter 67]. In [mccarthy 67], the source language consists of expressions, identifiers, and constants where the binary operator + is the only operator allowed. The source semantics is defined in terms of a state vector. The target language is defined in terms of a single address machine with an accumulator. The data structure map associates source identifiers with target memory locations, and source state vectors with target state vectors. The compiler design is correct if the outcome of an execution of any source program in any state is related to the accumulator contents after executing the compiled program. An argument must be made on which target memory locations are affected.

[painter 67] presents some of the same ideas as [mccarthy 67] with a larger example, an Algol-like source language. The complexity of the source language is about the same as the language Tiny which we considered in detail in Chapter 8.

In [london 71], London proves the correctness of a compiler for a subset of LISP. London informally states what the target code is for each source syntactic type that is input to the compiler, and then proceeds to show that the target code has the same effect as the source construct by a hand execution scheme. Because the source language was based on pure LISP (no assignments and no globals), the overall proof was based on structural induction over source syntax. The hand simulation technique sufficed for a simple example where the behavior of the source language is to return a single value.

In [boyer 77], a proof similar to that of [mccarthy 67] is done with the aid of a theorem prover. The paper also discusses the optimization phase of the compiler. The theorem prover deals with the theory of total recursive functions in a domain of axiomatically specified finitely constructable objects. In particular, it has knowledge about recursion and induction. The interpreters are written in a LISP-like language. The proof proceeds by structural induction. We could have as easily referred to this paper in the previous section, but no restriction was placed on the interpreter.

In [mazaher 81], the issue of compiler correctness is addressed where the specification languages investigated are VDL, Semanol, and high-level programming languages. The compiler is derived from a deterministic interpreter of the source language and the derivation process is proved to be semantics preserving. This is reminiscent of the work we mentioned earlier where the target semantics are derived from the source. An interpreter is transformed into a compiler by making the interpreter output code whenever a statechanging instruction is about to be executed. The objective is not analytic (proving a compiler correct), but rather, synthetic (deriving a correct compiler). It is important to note that the author came to the conclusion that (1) "interpreters written in a denotational style meet the goal of compiler generation better", and (2) "specification languages having facilities for defining abstract data types are more suitable for writing operational semantics." Both these remarks support the method proposed in this research where the specification language incorporates the concepts of abstract data types and denotational semantics.

In [mazaher 81], the operational semantics are restricted and marked (e.g., variables are marked as compile-time or run-time) to give it a denotational flavor. This also corresponds to the remark made earlier that in an operational semantics description, semantics of individual constructs or compile-time/run-time properties must be abstracted from a large algorithm. It appears that the semantics preserving transformation rules are proved correct using the usual interpreter equivalence method described earlier. This is a one-time task for each set of rules used.

Finally, even though this dissertation is primarily concerned with higher-order programming languages at the source level, it is relevant to mention in this section several papers involving rather low-level source languages. The papers address microcode correctness where microcode is used to implement a computer instruction set; computer hardware interprets the computer instruction set by executing microcode. Thus, the source specifies the computer instruction set and the target specifies the microcode. The interpreter approach has been more successful at this level because the source and target interpreters can be quite similar and the computer languages have simple grammatical specifications (a very flat hierarchy). Furthermore, at this level, the programmer usually perceives the programming language semantics in terms of a machine. References include [carter 78], [crocker 77], [dasgupta 84], [damm 84], [damm 85a], and [levy 84]. In particular, some recent (unpublished) work using the State Delta Verification System (SDVS) based on [marcus 84a] and [marcus 84b] has been done where a machine-checked proof of an implementation of about 120 computer instructions of the BBN C30 computer was completed. The language was implemented by about 1000 lines of microcode. This is the largest, real application of this technology known to the author.

9.3. Interfacing Denotational and Operational Semantics?

The opinion has been expressed in some of the literature cited above that (1) all programming languages must have an operational semantics definition, and/or (2) the lowest-level target must be specified with an operational semantics definition. Of course, this conflicts with our goals to have one verification approach for a multi-level (hierarchical) design, and at the same time have a verification approach that results in concise specifications, mirrors the informal design process, and results in small, independent verification tasks. One course of action is to employ a verification method that has the nice properties just mentioned for all levels of the design hierarchy, and then show that, say, a denotational semantics definition of the lowest level language can be implemented in the operational semantics definition of the same language.

The problem of showing that a denotational definition is complementary to an operational definition for the same language is discussed in [stoy 77], [mulmuley 85], and [schmidt 86]. Inclusive predicates are used. As mentioned earlier, this may require difficult existence proofs. The results in [mulmuley 85] offer hope that some of this can be made systematic and mechanized. Furthermore, to prove that a low-level operational semantics simulates a high-level semantics, the operational semantics must have properties of *faithfulness* and *termination* [schmidt 86]. An operational semantics is faithful if all evaluations of an expression denote the same value; in other words, it is well-defined or sound. It is terminating if you can guarantee forward progress to an answer; if two expressions denote the same value then there is a computation from one to the other.

Chapter 10

Conclusion

The goals of this project were to define a compiler design verification method that:

1. models the informal process of changing a representation and then determining whether the representation change is correct, and
2. is highly modular so that many verification tasks can be performed in parallel and can possibly be automated, and minor changes to specifications will have little affect on any existing verification.

In an attempt to meet these goals, the verification approach presented in this dissertation combines the concepts of interpretation between theories from mathematical logic, abstract data types, and denotational semantics. Theories which formally specify abstract data types are extended to allow higher order operators, domain constructors, and domain equations. The extended theories can be used to specify the denotational semantics and the abstract syntax of a programming language. An interpretation for the extended theories and criteria the interpretation must satisfy to be correct are defined. The interpretation is used as a formal specification of a compiler design. A mathematical proof that the interpretation is correct constitutes a compiler design verification.

The key characteristics of the correctness proof are:

1. the proof proceeds by structural induction on the source language syntax and the induction argument is implicitly handled by using the interpretation to translate the source theory.
2. the implementation of the source programming language syntax is treated in the same manner as the implementation of the source programming language semantics.

Chapter 10

Conclusion

The goals of this project were to define a compiler design verification method that:

1. models the informal process of changing a representation and then determining whether the representation change is correct, and
2. is highly modular so that many verification tasks can be performed in parallel and can possibly be automated, and minor changes to specifications will have little affect on any existing verification.

In an attempt to meet these goals, the verification approach presented in this dissertation combines the concepts of interpretation between theories from mathematical logic, abstract data types, and denotational semantics. Theories which formally specify abstract data types are extended to allow higher order operators, domain constructors, and domain equations. The extended theories can be used to specify the denotational semantics and the abstract syntax of a programming language. An interpretation for the extended theories and criteria the interpretation must satisfy to be correct are defined. The interpretation is used as a formal specification of a compiler design. A mathematical proof that the interpretation is correct constitutes a compiler design verification.

The key characteristics of the correctness proof are:

1. the proof proceeds by structural induction on the source language syntax and the induction argument is implicitly handled by using the interpretation to translate the source theory.
2. the implementation of the source programming language syntax is treated in the same manner as the implementation of the source programming language semantics.

3. a domain of source objects can be implemented as some subset of a domain of target objects and a source object can have two or more equivalent representations in the implementation.
4. the proof is systematically broken down into small, independent tasks that are amenable to automation; the proofs are done as target theory deductions, primarily using target semantic equations as rewrite rules.

The verification method is demonstrated with a series of examples in Chapter 8. While these examples contain constructs and domains one would typically see in real applications, the examples are relatively small compared to real applications. To scale up in size, computer assistance is needed. As mentioned above, any attempt at automating the verification tasks requires a significant investment of effort -- several person-years.

In order to apply the verification method, restrictions are placed on the interpretations allowed. A detailed discussion of the impact of these restrictions can be found in Sections 6.6 and 7.3. If the restrictions are not too limiting, then the method does satisfy the goals. If the restrictions need to be relaxed, then the verification approach proposed in this dissertation must be modified, if possible, to allow other types of interpretations. The latter requires more work in extending the mathematical framework presented in this dissertation.

This research also contains a review and comparison of other verification approaches. This is presented in Chapter 9. Our interpretation approach is most similar to the algebraic approach, but does result in a different proof organization. Rather than proving a map is a homomorphism, the proof in the interpretation approach consists of a translation step and a deduction step using the target theory. The interpretation approach is different from the twin machine approach in that the former organizes the proof by structural induction on the source language and the value; in other words, it is well-defined or sound. It is terminating if you can guarantee forward progress to an answer; if two expressions denote the same value then there is a computation from one to the other.

Chapter 10

Conclusion

The goals of this project were to define a compiler design verification method that:

1. models the informal process of changing a representation and then determining whether the representation change is correct. and
2. is highly modular so that many verification tasks can be performed in parallel and can possibly be automated. and minor changes to specifications will have little affect on any existing verification.

In an attempt to meet these goals, the verification approach presented in this dissertation combines the concepts of interpretation between theories from mathematical logic, abstract data types, and denotational semantics. Theories which formally specify abstract data types are extended to allow higher order operators, domain constructors, and domain equations. The extended theories can be used to specify the denotational semantics and the abstract syntax of a programming language. An interpretation for the extended theories and criteria the interpretation must satisfy to be correct are defined. The interpretation is used as a formal specification of a compiler design. A mathematical proof that the interpretation is correct constitutes a compiler design verification.

The key characteristics of the correctness proof are:

1. the proof proceeds by structural induction on the source language syntax and the induction argument is implicitly handled by using the interpretation to translate the source theory.
2. the implementation of the source programming language syntax is treated in the same manner as the implementation of the source programming language semantics.

3. a domain of source objects can be implemented as some subset of a domain of target objects and a source object can have two or more equivalent representations in the implementation.

4. the proof is systematically broken down into small, independent tasks that are amenable to automation; the proofs are done as target theory deductions, primarily using target semantic equations as rewrite rules.

The verification method is demonstrated with a series of examples in Chapter 8. While these examples contain constructs and domains one would typically see in real applications, the examples are relatively small compared to real applications. To scale up in size, computer assistance is needed. As mentioned above, any attempt at automating the verification tasks requires a significant investment of effort -- several person-years.

In order to apply the verification method, restrictions are placed on the interpretations allowed. A detailed discussion of the impact of these restrictions can be found in Sections 6.6 and 7.3. If the restrictions are not too limiting, then the method does satisfy the goals. If the restrictions need to be relaxed, then the verification approach proposed in this dissertation must be modified, if possible, to allow other types of interpretations. The latter requires more work in extending the mathematical framework presented in this dissertation.

This research also contains a review and comparison of other verification approaches. This is presented in Chapter 9. Our interpretation approach is most similar to the algebraic approach, but does result in a different proof organization. Rather than proving a map is a homomorphism, the proof in the interpretation approach consists of a translation step and a deduction step using the target theory. The interpretation approach is different from the twin machine approach in that the former organizes the proof by structural induction on the source language and the latter organizes the proof by induction on the computation steps of the machines. On the basis of the discussion in Chapter 9, it is the opinion of this author that the interpretation approach is better suited for dealing with high-level languages and the twin machine approach is better suited for dealing with low-level languages or for showing semantic definitions for the same low-level language are complementary. The algebraic approach has helped unify semantic definition methods and verification techniques. New research on algebraic semantics may result in further verification improvements. New semantic domains as abstract data types may simplify the specification and verification processes.

In summary, the original contributions of this research are:

1. interpretation between theories has been defined for theories that have been extended to have higher order operators, domains, and domain equations.
2. the application of interpretation between these extended theories to the compiler design correctness problem has been demonstrated.

This research's extension to interpretation between theories can be used for applications other than the compiler design problem. Other problems that can be formulated in terms of higher order abstract data types can make use of the verification method. New programming languages have incorporated the abstract data type concept (e.g., Ada), polymorphic data types (e.g., ML [gordon 79b]), or polymorphic higher order data types (e.g., HOPE [burstall 80]). Several functional programming languages (e.g. LISP) use higher order operations. Furthermore, there is a growing interest in the use of abstract data types to specify other applications (e.g., hardware, databases). [parsaye-ghomi 82] contains a good discussion of higher order abstract data types and some examples.

Finally, this research has identified issues for further study. An obvious proposal is to scale up the examples. What is not so obvious is the amount of effort that would be required to automate some of the verification tasks in order to tackle the larger problems. The specifications alone may take a year to write. Existing systems (e.g., LCF [gordon 79b], rewriting systems) should be investigated to see if they can be used or modified for use.

The issue of multi-level designs should also be addressed. Little work has been done to verify a compiler design with multiple levels of abstraction.

Examining larger examples and multi-level designs will identify deficiencies in the verification method. For example, such an examination will permit us to determine whether the interpretation restrictions are too limiting. If the present method needs generalization, it may lead to a redefinition of allowable domain subsets or quotients. The question of whether subsets and quotients of domains are themselves domains is basically open and is very hard. Other models for domains should be investigated in an attempt to solve these problems and, perhaps, simplify the discussion.

References

[andrews 82] D. Andrews and W. Henhapl.
Pascal.
Formal Specification and Software Development.
Prentice-Hall, Englewood Cliffs, N.J., 1982, pages 175 - 252.

[ashcroft 82] E. A. Ashcroft and W. W. Wadge.
Prescription for Semantics.
ACM Transactions on Programming Languages and Systems 4:194 - 283, 1982.

[barendregt 81] H. P. Barendregt.
The Lambda Calculus - Its Syntax and Semantics.
North-Holland Publishing Company, 1981.

[bjorner 80] Dines Bjorner.
Formal Description of Programming Concepts A Software Engineering Viewpoint.
In G. Goos and J. Hartmanis - P. Dembinski (editor), *Lecture Notes in Computer Science: Mathematical Foundations of Computer Science 1980 -- Proceedings of the 9th Symposium.* Springer-Verlag, May, 1980.

[boyer 77] Robert S. Boyer and J. Strother Moore.
A Computer Proof of the Correctness of a Simple Optimizing Compiler for Expressions.
Technical Report 5, Defense Technical Information Center, January, 1977.

[branquart 82] P. G. Branquart, G. Louis, and P. Wodon.
Lecture Notes in Computer Science 128: An Analytical Description of CHILL, the CCITT High Level Language.
Springer, Berlin, 1982.

[burstall 68] R. M. Burstall.
Proving Properties of Programs by Structural Induction.
Technical Report, University of Edinburgh, August, 1968.

[burstall 80] R. M. Burstall, D. B. MacQueen, D.T. Sannella.
Hope: An Experimental Applicative Language.
In *Conference Record of the 1980 LISP Conference.* 1980.

[carter 78] W. C. Carter, W. H. Joyner, and D. Brand.
Microprogram Verification Considered Necessary.
In *Proceedings of the National Computer Conference*, pages 657-664. AFIPS Press, Arlington, VA, 1978.

[chirica 76] L. M. Chirica.
Contributions to Compiler Correctness.
Technical Report UCLA ENG-7697, Computer Science Department, School of Engineering and Applied Science, University of California at Los Angeles, October, 1976.

[chirica 86] L. M. Chirica and D. F. Martin.
Toward Compiler Implementation Correctness Proofs.
ACM Transactions on Programming Languages and Systems 8(2):185-214, April, 1986.

[cohn 81] Avra Cohn.
The Equivalence of Two Semantic Definitions: a Case Study in LCF.
Internal Report CSR-76-81, University of Edinburgh, January, 1981.

[crocker 77] Stephen D. Crocker.
State Deltas: A Formalism for Representing Segments of Computation .
PhD thesis, University of California, 1977.

[damm 84] W. Damm.
An Axiomatization of Low-Level Parallelism in Microarchitectures.
In *The Seventeenth Annual Microprogramming Workshop*, pages 314-324. IEEE Computer Society, Computer Society Press, October - November, 1984.

[damm 85a] Werner Damm and Gert Dohmen.
Verification of Microprogrammed Computer Architectures.
In *Micro 18 Conference Proceedings*, pages 61-73. ACM, October, 1985.

[damm 85b] Werner Damm.
Design and Specification of Microprogrammed Computer Architectures.
In *Micro 18 Conference Proceedings*, pages 3 - 9. ACM, October, 1985.

[dasgupta 84] S. Dasgupta.
A Model of Clocked Micro-Architectures for Firmware Engineering and Design Automation Applications.
In *The Seventeenth Annual Microprogramming Workshop*, pages 298-308. IEEE Computer Society, Computer Society Press, October - November, 1984.

[despeyroux 86] Joelle Despeyroux.
Proof of Translation in Natural Semantics.
In *Symposium on Logic in Computer Science*, pages 193-205. IEEE Computer Society, Computer Society Press, June, 1986.

[donzeau-gouge 80]
V. Donzeau-Gouge.
On the Formal Description of Ada.
Lecture Notes in Computer Science 94: Semantics-Directed Compiler Generation.
Springer, Berlin, 1980.

[dybjer 83] Peter Dybjer.
Category-Theoretic Logics and Algebras of Programs.
PhD thesis, Chalmers University of Technology, University of Gothenburg, Department of Computer Sciences, 1983.

[enderton 72] Herbert B. Enderton.
A Mathematical Introduction to Logic.
Academic Press, 1972.

[goguen 76] J. A. Goguen, J. W. Thatcher, E. G. Wagner.
An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types.
Research Report RC 6487 (#26817), IBM Thomas J. Watson Research Center, October, 1976.

[good 75] Donald I. Good, Ralph L. London, W. W. Bledsoe.
An Interactive Program Verification System.
IEEE Transactions on Software Engineering :59 - 67, March, 1975.

[gordon 73] M. J. C. Gordon.
Models of Pure LISP.
Experimental Programming Reports 31, Machine Intelligence Dept., Edinburgh University, 1973.

[gordon 79a] Michael J. C. Gordon.
The Denotational Description of Programming Languages.
Springer-Verlag, 1979.

[gordon 79b] Michael Gordon, Robin Milner, and Christopher Wadsworth.
Lecture Notes in Computer Science 78: Edinburgh LCF.
Springer-Verlag, Berlin, 1979.

[guttag 78] John V. Guttag, Ellis Horowitz, and David R. Musser.
Abstract Data Types and Software Validation.
Communications of the ACM 21(12), December, 1978.

[henhapl 82] W. Henhapl and C. Jones.
ALGOL 60.
Formal Specification and Software Development.
Prentice-Hall, Englewood Cliffs, N.J., 1982.

[hoare 72] C.A.R. Hoare.
Proof of Correctness of Data Representations.
Acta Informatica :271-281, February, 1972.

[johnston 71] J. B. Johnston.
The Contour Model of Block Structured Processes.
In *Proceedings of the ACM Symposium on Data Structures in Programming Languages.* February, 1971.

[kini 82] V. Kini, D. Martin, and A. Stoughton.
Testing the INRIA Ada Formal Definition: The USC-ISI Formal Semantics Project.
In *Proceedings ADATec Meeting.* 1982.

[levy 84] Beth Levy.
Microcode Verification Using SDVS: The Method and a Case Study.
In *The Seventeenth Annual Microprogramming Workshop*, pages 234-245. IEEE Computer Society, October - November, 1984.

[london 71] Ralph L. London.
Correctness of Two Compilers for a LISP Subset.
Technical Report CS 240, Stanford University, Computer Science Department, October, 1971.

[lucas 68] P. Lucas.
Two Constructive Realizations of the Block Concept and Their Equivalence.
Technical Report TR-25. 085, IBM Laboratory, 1968.

[lucas 70] P. Lucas, P. Lauer, and H. Stigleitner.
Method and Notation for the Formal Definition of Programming Languages.
Technical Report TR-25. 085, IBM Laboratory, February, 1970.

[lynn 78] Donald Scott Lynn.
Interactive Compiler Proving Using Hoare Proof Rules..
Technical Report ISI/RR-78-70, Information Sciences Institute, January, 1978.

[manna 72] Z. Manna, S. Ness, J. Vuillemin.
Inductive Methods for Proving Properties of Programs.
In *Proceedings of an ACM Conference on Proving Assertions about Programs*, pages 27-50. Association for Computing Machinery, SIGPLAN and SIGACT, January, 1972.

[marcus 84a] L. Marcus, S. D. Crocker, and J. R. Landauer.
SDVS: A System for Verifying Microcode Correctness.
In *The Seventeenth Annual Microprogramming Workshop*, pages 246-257. IEEE Computer Society, Computer Society Press, October - November, 1984.

[marcus 84b] L. Marcus and J. V. Cook.
SDVS User Manual.
ATR 84(8478)-1, The Aerospace Corporation, 1984.

[mazaher 81] Shahrzade Mazaher.
An Approach to Compiler Correctness.
PhD thesis, UCLA, 1981.

[mccarthy 67] J. McCarthy and J. Painter.
Correctness of a Compiler for Arithmetic Expressions.
In *Mathematical Aspects of Computer Science, Vol. 19: Proceedings of Symposium in Applied Mathematics*, pages 33-41. March, 1967.

[mcgowan 72] Clement L. McGowan.
An Inductive Proof Technique for Interpreter Equivalence.
Formal Semantics of Programming Languages.
Prentice-Hall, 1972, pages 139-147.

[milne 72] R. E. Milne.
The Mathematical Semantics of Algol68 .
Technical Report, Programming Research Group, Oxford University, 1972.

[milne 76] Robert Milne.
A Theory of Programming Language Semantics.
Chapman and Hall Ltd., 1976.

[milne 83] George J. Milne.
The Correctness of a Simple Silicon Compiler.
Internal Report CSR-127-83, University of Edinburgh, January, 1983.

[milner 72] R. Milner and R. Weyhrauch.
Proving Compiler Correctness in a Mechanized Logic.
Machine Intelligence 7.
Halsted Press, 1972, pages 51-70, Chapter 3.

[morris 73] L. Morris.
Advice on Structuring Compilers and Proving Them Correct.
In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 144-152. ACM, 1973.

[mosses 74] P. D. Mosses.
The Mathematical Semantics of Algol60.
Tech Monograph PRG12, Programming Research Group, Oxford University, 1974.

[mosses 75] Peter David Mosses.
Mathematical Semantics and Compiler Generation.
Master's thesis, University of Oxford, April, 1975.

[mosses 80] Peter Mosses.
A Constructive Approach to Compiler Correctness.
In G. Goos and J. Hartmanis (editor), *Lecture Notes in Computer Science: Automata, Languages and Programming Seventh Colloquium*, pages 449-469. Springer-Verlag , July, 1980.

[muchnick 82] S. S. Muchnick and U. Pleban.
A Semantic Comparison of LISP and Scheme.
In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 56 - 64. 1982.

[mulmuley 85] Ketan Mulmuley.
Full Abstraction and Semantic Equivalence.
PhD thesis, Carnegie-Mellon University, Department of Computer Science, 1985.

[odonnell 82] Michael J. O'Donnell.
A Critique of the Foundations of Hoare Style Programming Logics.
Communications of the ACM 25(12):927-935, December, 1982.

[orejas 84] Fernando Orejas.
Even More on Advice on Structuring Compilers and Proving them Correct: Changing an Arrow.
ACM Special Interest Group on Programming Languages. 19(6):pp. 82-84, 1984.

[painter 67] James Allan Painter.
Semantic Correctness of a Compiler.
PhD thesis, Stanford University, Department of Computer Science, February, 1967.

[parsaye-ghomi 82]
Kamran Parsaye-Ghomi.
Higher Order Abstract Data Types.
Technical Report CSD-820112, UCLA, Computer Science Department, January, 1982.

[polak 80] Wolfgang Heinz Polak.
Theory of Compiler Specification and Verification.
Technical Report STAN-CS-80-802, Stanford University, Computer Science Department, May, 1980.

[reynolds 72] John C. Reynolds.
Definitional Interpreters for Higher-Order Programming Languages.
In J. E. Fenstad (editor), *Proceedings ACM National Conference*, pages 717-740. ACM, North-Holland Publishing Company, Amsterdam, December, 1972.

[reynolds 74] J. C. Reynolds.
On the Relation Between Direct and Continuation Semantics.
In *Lecture Notes in Computer Science, Proceedings of the 2nd Colloquium on Automatic Languages and Programming*, pages 141-156. Springer-Verlag, 1974.

[royer 86] Veronique Royer.
Transformations of Denotational Semantics in Semantics Directed Compiler Generation.
In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 68-73. ACM, June, 1986.

[schmidt 86] D. A. Schmidt.
Denotational Semantics.
Allyn and Bacon, Inc., 1986.

[scott 76] D. Scott.
Data Types as Lattices.
Siam J. Comput 5(3):522-587, September, 1976.

[shoenfield 67] Joseph R. Shoenfield.
Mathematical Logic.
Addison-Wesley Publishing Company, 1967.

[stanford 79] Stanford Verification Group.
Stanford Pascal Verifier User Manual.
Technical Report 11, Stanford University, 1979.

[stoy 77] Joseph E. Stoy.
Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.
MIT Press, 1977.

[tennent 73] R. D. Tennent.
Mathematical Semantics of SNOBOL4.
In *Proceedings of the 1st ACM Symposium on Principles of Programming Language*, pages 95 - 107. ACM, Boston, 1973.

[tennent 77] R. D. Tennent.
A Denotational Definition of the Programming Language Pascal.
Technical Report 77-47, Department of Computing and Information Sciences, Queen's University, Kingston, Ontario, 1977.

[thatcher 79] J. W. Thatcher.
More Advice on Structuring Compilers and Proving Them Correct.
In J. E. Fenstad (editor), *Lecture Notes in Computer Science*, pages 596-615. ACM, North-Holland Publishing Company, Amsterdam, July, 1979.

[turner 79] D. A. Turner.
A New Implementation Technique for Applicative Languages.
Software Practice and Experience 9(1):31-49, January, 1979.

[wadsworth 78] C. Wadsworth.
AI/CS PG Course Notes.
1978.

[wand 80] Mitchell Wand.
First-Order Identities as a Defining Language.
Acta Informatica 27(14):337-357, 1980.

[wand 82a] Mitchell Wand.
Specifications, Models, and Implementations of Data Abstractions.
Theoretical Computer Science 27(20):3-32, 1982.

[wand 82b] Mitchell Wand.
Deriving Target Code as a Representation of Continuation Semantics.
ACM Transactions on Programming Languages and Systems 4(3):496-517, July, 1982.

[wegner 70] P. Wegner.
Programming Language Semantics.
In *Proceedings of Courant Institute Computer Science Symposium*, pages 149 - 242. New York, September, 1970.

[wegner 72] P. Wegner.
The Vienna Definition Language.
Computing Surveys ACM 4:1:5 - 62, March, 1972.

Appendix A

Interpretation Between Theories For Predicate Calculus

The methodology presented in this dissertation for specifying, implementing and verifying abstract data types is founded on mathematical logic, in particular, "interpretation between theories". This methodology is used in the development of correctness criteria for compilers. This appendix presents some background material dealing with mathematical logic. It was primarily extracted and summarized from [shoenfield 67] and [enderton 72].

In any proof there are mathematical laws, called *axioms*, that are accepted without proof. Other mathematical laws, called *theorems*, are proved from the axioms. An axiom may be viewed as a sentence (i.e., in terms of its syntax) or as the meaning of a sentence (i.e. in terms of its semantics or structure). If the language used for expressing axioms is well-defined, then the syntax of each axiom will reflect its meaning. Thus, we can study axioms and the theorems derived from the axioms by studying the syntax of the sentences expressing them.

A *formal system* permits syntactic investigations of axioms and theorems. Specifically, a formal system consists of:

1. a language
2. axioms
3. rules of inference

These items are defined below.

A *symbol* is an "atomic object;" no symbol is a sequence of other symbols. An *expression* is any finite sequence of symbols. A *language* of a formal system is specified by

1. specifying the symbols
2. specifying the *formulas* which are grammatically correct expressions of the language

The *axioms* are formulas expressed in the language of the formal system. *Rules of inference*, the third part of a formal system, provide a means to derive theorems from the axioms. "Each rule of inference states that under certain conditions, one formula, called the *conclusion* of the rule, can be *inferred* from certain other formulas, called the *hypotheses* of the rule" [shoenfield 67]. If H denotes the hypotheses and C the conclusion, then the rule of inference is typically written

$$\frac{H}{C}$$

The inferred formula is a *theorem* if the hypotheses are theorems. All axioms in a formal system are theorems in the formal system. If A is a theorem of a formal system F, then it is written as $\vdash_F A$ where the subscript F is omitted if the context is unambiguous. A *proof* in a formal system is the finite sequence of formulas obtained by applying rules of inference. "If A is the last formula in a proof P, we say that P is a proof of A" [shoenfield 67].

A *first-order theory (or theory)*, call it T, is a class of formal systems. The language of T is a *first-order-language*; call it L. L has two types of symbols: logical symbols and nonlogical symbols (or parameters). The *logical symbols* are:

1. parentheses: (,)
2. sentential connective symbols: $\Rightarrow$, $\neg$ (or alternatively, $\neg$, $\vee$, $\exists$)
3. variables

The *nonlogical symbols* are:

1. quantifier symbol: $\forall$
2. n-place predicate symbols where $n \geq 1$
3. n-place function symbols where $n \geq 1$
4. constant symbols

The meaning of the logical symbols is fixed, but the nonlogical symbols are open to interpretation.

Formulas in L are defined using terms and atomic formulas. A *term* is either:

1. a variable, or
2. $fu_1...u_n$ where $u_1...u_n$ are terms and f is an n-place function symbol

An *atomic formula* is an expression of the form $pt_1...t_n$ where p is an n-place predicate symbol and $t_1...t_n$ are terms. A *well-formed formula* (or *formula*) is one of the following:

1. an atomic formula
2. $\neg P$, $P \Rightarrow Q$, and $\forall v$: P, where P and Q are formulas and v is a variable

Depending on the axioms and rules of inference selected for T, T may or may not have the useful properties of soundness and completeness. We will discuss why these properties are desirable and why soundness is necessary for correctness proofs. Then, we will conclude this section with a discussion about interpretation between theories where we describe how to show one theory is as powerful as another and how soundness permits us to tackle this problem.

Informally, if T is sound then any theorem of T will be in some sense true. If T is complete, any true formula expressed in L will be a theorem of T; i.e., T is powerful enough to derive all true formulas of the language. To express these properties more formally we will need some definitions.

A *structure*, A, for the language L is a function whose domain is the set of parameters of L such that

1. A assigns to $\forall$ a nonempty set $|A|$, called the *universe* or *carrier* of A.
2. A assigns to each n-place predicate symbol P an n-ary relation $P^A \subseteq |A|^n$; P^A is a set of n-tuples of members of the universe.
3. A assigns to each constant symbol C a member of C^A of the universe $|A|$.
4. A assigns to each n-place function symbol f an n-ary operation f^A on $|A|$; i.e., f^A: $|A|^n \rightarrow |A|$.

If α is a well-formed formula it has a set $fv(\alpha)$ of free variables. This set is defined inductively by:

1. $fv(x) = \{x\}$, where x is a variable
2. $fv(gt_1...t_n) = fv(t_1) \cup ... \cup fv(t_n)$, where g is an n-place function or predicate symbol and $t_1, ..., t_n$ are terms
3. $fv(\neg\alpha) = fv(\alpha)$
4. $fv(\alpha \Rightarrow \beta) = fv(\alpha) \cup fv(\beta)$
5. $fv(\forall v{:}\alpha) = fv(\alpha) - \{v\}$

Let α be a well-formed formula, A a structure, and s: $V \rightarrow |A|$ a function from the set V of all variables into the universe $|A|$ of A. Call s the environment or state. A *satisfies* α *with s*, $\models_A \alpha[s]$, if and only if the translation of α determined by A, where the variable x is translated s(x) wherever it occurs free, is true. A *is a model of* α (or α is valid in A), $\models_A \alpha$, if and only if A satisfies α with every environment s. This can be written as

$$\models_A \alpha \text{ iff } (\forall s)\ (\models_A \alpha[s])$$

α *is valid*, $\models\alpha$, if and only if for every structure A and every environment s, A satisfies α with s. This can be written

$$\models\alpha \text{ iff } (\forall A)(\forall s)\ (\models_A \alpha[s])$$

Let Γ be a set of well-formed formulas and α a well-formed formula. Then Γ *logically implies* α (α *is a logical consequence of* Γ), $\Gamma\models\alpha$, if and only if for every structure A for L and every environment s such that A satisfies every member of Γ with s, A also satisfies α with s. Writing this in mathematical notation, we have

$$\Gamma\models\alpha \text{ iff } (\forall A)\ (\forall s)\ (\models_A \Gamma[s] \Rightarrow \models_A \alpha[s])$$

Let Λ be the set of valid formulas called *logical axioms* for first-order theories (these are defined in [enderton 72] and [shoenfield 67]) and let Γ be a set of formulas called non-logical axioms. A *is a model of theory T* if and only if all the formulas in Γ are valid in A.

If α *is a theorem of* Γ (α is a theorem of a first-order theory assuming formulas Γ are also theorems), then the sequence of formulas that records how α was obtained from $\Gamma\cup\Lambda$ with the rule(s) of inference for first-order theories is called a *deduction* or *proof* of α from Γ. α is a theorem of Γ is written $\Gamma\vdash\alpha$.

For first-order theories, the Soundness Theorem states if $\Gamma\vdash\alpha$ then $\Gamma\models\alpha$. For first-order theories, the Completeness Theorem states if $\Gamma\models\alpha$ then $\Gamma\vdash\alpha$.

Recently, new languages and rules for reasoning about computer programs have been proposed. Several of the proposed formal systems have not been sound and thus, the correctness proofs have not been based on sound reasoning. "If a formal system is to provide a satisfactory foundation for actual reasoning, the methods of proof should be intuitively correct, not just symbol manipulation tricks that fortuitously produce true theorems at the end [odonnell 82]." Any theorem proved in a theory should also be a logical consequence of the theory; the proofs should be based on sound reasoning. To show soundness, it must be shown that the axioms are valid and any formulas obtained by the rules of inference are logical consequences of the hypotheses. However, it is not always possible for many useful theories to satisy the completeness property (e.g., number theory). It would be nice to know we can always find a proof for valid formulas, but we frequently have to be satisfied knowing that if we did find a proof of formula α, α is a logical consequence of the theory.

Interpretation between theories is a useful concept in mathematical logic. Given two theories, T_1 and T_2, it is possible to show that T_2 is as powerful (precise) as T_1. If T_1 and T_2 are in the same language and $T_1 \subseteq T_2$ then it is obvious that T_2 is as powerful as T_1. The interesting problems occur when the theories are in different languages. If the theories are in different languages and T_2 is as powerful as T_1 then there must exist a translation from the language of T_1 to the language of T_2 (i.e., the image of one theory is contained in another).

Let L_1 be the language of T_1 and L_2 be the language of T_2. An *interpretation* π of L_1 into T_2 is a function on the set of nonlogical symbols of L_1 such that

1. π assigns to $\forall$ a formula $\pi_\forall$ of L_2 in which at most the variable v_1 occurs free, such that

   (i) $T_2 \models \exists v_1 \pi_\forall$

2. π assigns to each n-place predicate symbol P a formula π_P of L_2 in which at most the variables $v_1, \ldots, v_n$ occur free

3. π assigns to each n-place function symbol f a formula π_f of L_2 in which at most $v_1, \ldots, v_n, v_{n+1}$ occur free, such that

   (ii) $T_2 \models \forall v_1 \ldots \forall v_n (\pi_\forall(v_1) \Rightarrow \ldots \Rightarrow \pi_\forall(v_n)$

$$\Rightarrow \exists x(\pi_\forall(x) \wedge \forall v_{n+1}(\pi_f v_1, \ldots, v_n = v_{n+1} \text{ iff } v_{n+1} = x)))$$

The idea behind (i) is that in any model of T_2, the formula $\pi_\forall$ should define a nonempty set to be used as the universe of an L_1-structure. The idea behind (ii) is that in any model of T_2, π_f defines a function on the universe defined by $\pi_\forall$.

The interpretation π can be extended to formulas. Any formula α of L_1 can be translated to a formula $\pi(\alpha)$ in the following manner:

1. if α is an atomic formula $pt_1 \ldots t_i \ldots t_n$, $1 \le i \le n$, and none of the t_i are function symbols then $\pi(pt_1 \ldots t_i \ldots t_n) = \pi_p t_1 \ldots t_i \ldots t_n$
2. if α is an atomic formula $pt_1 \ldots t_i \ldots t_n$, $1 \le i \le n$, and t_i is the rightmost function symbol then $\pi(pt_1 \ldots t_i \ldots t_n) = \forall y(\pi_{t_i}\ t_{i+1} \ldots t_n = y \Rightarrow \pi(pt_1 \ldots t_{i-1} y))$.
   (N.B., $pt_1 \ldots t_i \ldots t_n$ is logically equivalent to $\forall y\ (t_i t_{i+1} \ldots t_n = y \Rightarrow pt_1 \ldots t_{i-1} y)$)
3. for nonatomic formulas, $\pi(\neg\alpha) = \neg\pi(\alpha)$, $\pi(\alpha \Rightarrow \beta) = \pi(\alpha) \Rightarrow \pi(\beta)$, and $\pi(\forall v\colon \alpha) = \forall v\ (\pi_\forall(v) \Rightarrow \pi(\alpha))$.

If π is an interpretation and B is a model of T_2 then the following is a simple way to extract from B a structure B^π for L_1:

the universe of B^π, $|B^\pi|$:
$|B^\pi|$ = the set defined in B by $\pi_\forall$

the n-ary relation P^{B^π} assigned to each n-place predicate symbol P:
P^{B^π} = the relation defined in B by π_P, restricted to $|B^\pi|$

the n-ary operation f^{B^π} assigned to each n-place function symbol f:
$f^{B^\pi}(a_1, \ldots, a_n)$ = the unique b such that $\models_B \pi_f(a_1, \ldots, a_n) = b$,
where $a_1, \ldots, a_n$ are in $|B^\pi|$

If α is a formula in L_1 that is true in every structure B^π obtainable from a model B of T_2 then the translation of α, $\pi(\alpha)$, is true in model B with the same environment. Conversely, if $\pi(\alpha)$ is true in model B with the environment restricted to $|B^\pi|$ then α is true in the structure B^π. This means that the intuitive notion of interpretation of formulas is defined correctly. This property is stated in the following lemma.

Lemma 1: Let π be an interpretation of L_1 into T_2 and let B be a model of T_2. For any formula α of L_1 and any map s of the variables into $|B^\pi|$, $(\models_{B^\pi} \alpha[s])$ iff $(\models_B \pi(\alpha)[s])$

Proof: We will use structural induction on α.

Basis: α is an atomic formula $pt_1...t_i...t_n$, $1 \le i \le n$. We will use induction on the number of places at which function symbols occur in the atomic formula.

If none of the t_i are function symbols then $\models_B \pi_p t_1...t_n[s]$ iff $\models_{B^\pi} pt_1...t_n[s]$ because the variables t_i in each formula are assigned the same values and B^π assigns π_p to the predicate p.

If t_i is the rightmost function symbol then

$\models_B \pi(pt_1...t_i...t_n)[s]$

iff $\models_B \forall y(\pi_{t_i} t_{i+1}...t_n = y \Rightarrow \pi(pt_1...t_{i-1}y))[s]$ (definition of π)

iff $\models_B \pi(pt_1...t_{i-1}y)[s(b/y)]$ (where b = the unique b such that $\models_B \pi_{t_i} t_{i+1}...t_n[s] = b$)

iff $\models_{B^\pi} pt_1...t_{i-1}y\ [s(b/y)]$ (induction hypothesis)

iff $\models_{B^\pi} pt_1...t_{i-1}\pi_{t_i} t_{i+1}...t_n[s]$ (substitution lemma: $\models \alpha^x_t[s]$ iff $\models \alpha[s(s(t)/x)]$)

iff $\models_{B^\pi} pt_1...t_i...t_n\ [s]$ (definition of B^π)

Induction Step: α is a nonatomic formula.

Case 1: if α is $\neg\phi$ then

$\models_B \pi(\neg\phi)[s]$

iff $\models_B \neg\pi(\phi)[s]$ (definition of π)

iff $\models_{B^\pi} \neg\phi[s]$ (induction hypothesis)

Case 2: if α is $\phi\Rightarrow\gamma$ then

$\models_B \pi(\phi\Rightarrow\gamma)[s]$

iff $\models_B \pi(\phi)\Rightarrow\pi(\gamma)[s]$ (definition of π)

iff $\models_{B^\pi} \phi\Rightarrow\gamma[s]$ (induction hypothesis)

Case 3: if α is $\forall v$: ϕ then

$\models_B \pi(\forall v\colon \phi)[s]$

iff $\models_B \forall v\ (\pi_V(v)\Rightarrow\pi(\phi))[s]$ (definition of π)

iff $\models_{B^\pi} \forall v\ (\pi_V(v)\Rightarrow\phi[s]$ (induction hypothesis)

iff $\models_{B^\pi} \forall v\colon \phi[s]$ (definition of B^π)

An *interpretation* π *of a theory* T_1 *into a theory* T_2 is an interpretation π of the language L_1 of T_1 into T_2 such that if α is a valid L_1-sentence (i.e., $T_1 \models \alpha$) then $\pi(\alpha)$ is a valid L_2-sentence (i.e., $T_2 \models \pi(\alpha)$).

We can prove that π is an interpretation of T_1 into T_2 (T_2 is as powerful as T_1) if T_1 and T_2 possess certain properties. As described above, if T is sound and α is a theorem in T (i.e., $T \vdash \alpha$) then α is a valid L-sentence (i.e., $T \models \alpha$). If T is complete and α is a valid L-sentence (i.e., $T \models \alpha$) then α is a theorem in T (i.e., $T \vdash \alpha$).

Case 1: Say T_2 is sound and complete. If π is an interpretation of T_1 into T_2 the translation of every valid L_1-sentence is deducible in T_2 and valid in T_2. That is,

$$T_1 \models \alpha \xrightarrow{T_2 \text{ complete}} T_2 \vdash \pi(\alpha) \xrightarrow{T_2 \text{ sound}} T_2 \models \pi(\alpha)$$

Case 2: Say both T_1 and T_2 are sound and T_2 is complete. If π is an interpretation of T_1 into T_2 the translation of every theorem of T_1 will be valid in T_2. That is,

$$T_1 \vdash \alpha \xrightarrow{T_1 \text{ sound}} T_1 \models \alpha \xrightarrow{T_2 \text{ complete}} T_2 \vdash \pi(\alpha) \xrightarrow{T_2 \text{ sound}} T_2 \models \pi(\alpha)$$

Case 3: Say both T_1 and T_2 are sound and complete. If π is an interpretation of T_1 into T_2 every valid L_1-sentence will be a theorem of T_1 and its translation will be deducible and valid in T_2. That is,

$$T_1 \vdash \alpha \xleftrightarrow[\text{and complete}]{T_1 \text{ sound}} T_1 \models \alpha \xrightarrow{T_2 \text{ complete}} T_2 \vdash \pi(\alpha) \xrightarrow{T_2 \text{ sound}} T_2 \models \pi(\alpha)$$

By case 3, if T_1 and T_2 are both sound and complete, π is an interpretation of T_1 into T_2 if the translation of the axioms and rules of T_1 are deducible in T_2. In practice, T_1 and T_2 may not be complete. If T_2 is not complete we may not be able to deduce $\pi(\alpha)$ even if it is true. But, since T_2 is sound we know that if we do deduce

$\pi(\alpha)$ (even though T_2 is not complete) we know π is an interpretation of T_1 into T_2; we may not be able to prove some correct interpretations, but we never approve of incorrect interpretations. On the other hand, if T_1 is not complete, all the valid L_1-sentences are not necessarily deducible in T_1. Therefore, it is conceivable that even if the translation of axioms and rules of T_1 are deducible in T_2, the translation of some valid L_1-sentences may still not be valid in T_2. This means π may not be a correct interpretation of T_1 in T_2. The situation can be remedied by restricting T_1 such that the only L_1-sentences allowed in T_1 are the ones generated by the axioms and rules of inference in T_1 (i.e., T_1 is closed under deduction).

Appendix B

Wand's Extension to Interpretation Between Theories and its Application to Abstract Data Types

B.1. Abstract Data Types

Abstraction is a method used to reduce the amount of detail considered at any one time. Software and hardware implementations contain an enormous amount of detail, more than can be comprehended at any one time. By abstracting (or separating) attributes of an implementation that are relevant in a given context from those that are not, the amount of detail that must be handled during the design and verification of software and hardware becomes tractable [guttag 78].

An *abstract data type* (or data abstraction) is a mechanism for isolating attributes or properties of the structural relationship present within data. Computer programmers use abstract data types for designing software in a structured or top-down manner. By utilizing abstract data types in the algorithm designed to solve a problem, the software designer is not forced to use a given set of data types, and thus, not initially bogged down with implementation details: the problem is solved more simply or elegantly with data structured to fit the problem domain. Implementation details can be postponed and different implementations can be tried until one is found that meets the efficiency and computer constraints. For example, a stack is a data abstraction commonly used in software. If a stack is not a data type in the programming language used it could be implemented with other data types. such as an array and a pointer or a linked list. There may be many levels of abstraction between the most abstract level and the lowest implementation level considered.

The definition of abstract data type evolved from a description of the organization of data to a specification of operations allowable on objects belonging to the data type. In the early days of software development, the definition of a data type consisted of a particular implemented representation of a set of values. As more software was developed, the advantages of abstracting conceptual properties of data from implementation strategies became apparent. This is analogous to an earlier phase of abstract programming techniques and information hiding in which high level programming languages and compilers used to translate them were developed to alleviate the difficulty in writing and verifying assembly language programs [parsaye-ghomi 82].

Today's high level programming languages incorporate "basic" data types (e.g., arrays, integers, lists) and some languages provide a means for the programmer to define new data types. In fact, a programming language in its entirety can be considered an abstract data type. This is discussed in the dissertation.

B.2. Abstract Data Type Specification

Different languages have been developed to specify the operations of a data type. In [hoare 72] an abstract data type consists of a set of "abstract" values and some functions on those abstract values. The specification of an operation is given by two predicates called the *precondition* and the *postcondition*. The truth of the precondition before the application of an operation implies the truth of the postcondition after such an application, provided the operation terminates. This is expressed as a formula of the form P{A}Q where P is the precondition, Q is the postcondition and A is the operation. For example, consider the specification of the data type stack that can contain at most 100 integers. The stack has three operations: (1) INIT initializes a new stack and sets its length to zero, (2) PUSH takes a stack and an integer as arguments and if the length of the stack is less than 100 the integer is stored on top of the stack and the length of the stack is incremented by one, and (3) POP takes a stack as an argument and if the length of that stack is greater than zero the top element of the stack is removed and the length of the stack is decreased by one. The abstract values of the stack are represented by a sequence of integers enclosed in brackets. The rightmost integer in the sequence represents the top of the stack. The formulas are as follows:

1. **true** $\{\mathrm{INIT}(s)\}\ s = \langle\ \rangle\ \&\ \mathrm{LENGTH}(s) = 0$
2. $\mathrm{LENGTH}(s) < 100\ \&\ s = \langle x_1, \ldots, x_i \rangle\ \&\ i = \mathrm{LENGTH}(s)\ \{\mathrm{PUSH}(s,n)\}\ s = \langle x_1, \ldots, x_i, n \rangle\ \&\ \mathrm{LENGTH}(s) = i + 1$
3. $\mathrm{LENGTH}(s) > 0\ \&\ s = \langle x_1, \ldots, x_i \rangle\ \&\ i = \mathrm{LENGTH}(s)\ \{\mathrm{POP}(s)\}\ s = \langle x_1, \ldots x_{i-1} \rangle\ \&\ \mathrm{LENGTH}(s) = i - 1$

Another approach to abstract data type specification is the algebraic approach. [goguen 76, guttag 78] It further removes one from considering implementation strategies by eliminating representations for abstract values. The approach is to describe something without being committed to a particular representation. For example, in the theory of programming languages, *abstract syntax* considers syntactic structure, independently of whether it is represented by derivation trees, parenthesized expressions, indented program text, canonical parses, etc. [goguen 76]. Algebraic isomorphism provides a means to define abstraction in this way.

In the algebraic approach, an abstract data type is defined as a collection of sorts, operators and axioms. The *sorts* denote the various types of objects which are required for the data type. The operands and results of the operators are objects whose types make up the sorts. The axioms, usually written as algebraic equations, define the results of various combinations of operators applied to various operands. The operands may be variables of a specified type. The example given above for a bounded stack of integers of size 100 is specified as an algebraic presentation below:

1. sorts:

    stk
    int
    error
    bool

2. operators:

    INIT: → stk
    PUSH: stk X int → stk ∪ error
    POP: stk → stk ∪ error
    LENGTH: stk → int
    +: int × int → int
    =: int × int → bool

3. variables

    s: stk
    n: int
    ERROR: error

4. axioms:

    POP (PUSH (s,n)) = s
    LENGTH (INIT) =0
    LENGTH (PUSH (s,n)) = LENGTH (s) + 1
    PUSH (s,n) = ERROR, if LENGTH (s) = 100
    POP (INIT) = ERROR, if LENGTH (s) = 0

Finally, we will consider a third specification language for abstract data types called a many-sorted first-order Dynamic Logic (DLP) as described in [wand 82a]. DLP is defined as a language of a formal system. This language subsumes the first two languages discussed in this section; DLP has formulas of the form P{A}Q and it also has "typed" or "sorted" operators.

Wand postulates that any specification language for abstract data types can be reformulated in terms of a language of a formal system and that the methodology for proving correctness is largely independent of the specification languages used. We discuss DLP in detail because we wish to summarize the discussion in [wand 82a] which provides a basis for the definition of compiler correctness. We will not use DLP in the examples of compiler specification correctness proofs, but will present another language suited to that application.

The specification of an abstract data type is a set of sentences or formulas in some logical language, in this case DLP. The operations of the abstract data type are nonlogical symbols of the logical language and appear in the formulas. The formulas are formal statements of the properties of the abstract data type. The formulas are true or false given a particular structure for the language of the data type.

The nonlogical symbols of a first-order language are:

1. quantifier symbol
2. n-place predicate symbols
3. n-place function symbols
4. constant symbols

DLP extends this language by adding:

1. sort symbols
2. procedure symbols

Furthermore, all the symbols in DLP have a *signature* which identifies the "type" of the symbols. Each n-place function symbol has a signature $<\sigma_1,\ldots,\sigma_n>\to\sigma$ where $n \geq 0$ and $\sigma_1,\ldots,\sigma_n, \sigma$ are sort symbols. A constant symbol and a quantifier symbol are treated as a 0-place function symbol. Each n-place predicate symbol has a signature $<\sigma_1,\ldots,\sigma_n>$ where $n \geq 0$, and $\sigma_1,\ldots\sigma_n$ are sort symbols. Each procedure symbol has a signature $<\sigma_1,\ldots,\sigma_n>\to<\tau_1,\ldots,\tau_m>$ where $n, m \geq 0$ and $\sigma_1,\ldots,\sigma_n,\tau_1,\ldots,\tau_m$ are sort symbols. For each sort symbol σ, there are two distinguished procedure symbols: $ASSIGN_\sigma$ with signature $<\sigma>\to<\sigma>$, and $FORALL_\sigma$ with signature $\to<\sigma>$ (i.e., $FORALL_\sigma$ is a constant).[11]Each individual variable symbol has a sort σ where σ is a sort symbol.

Terms and atomic formulas are constructed as in first-order languages with the additional constraint that the sorts must "agree". This is described in the following definitions. A *term* is either:

1. an individual variable symbol of sort σ, or
2. $ft_1\ldots t_n$ where f is an n-place function symbol of signature $<\sigma_1,\ldots,\sigma_n>\to\sigma$ and $t_1\ldots t_n$ are terms of sorts $\sigma_1,\ldots,\sigma_n$.

An *atomic formula* is an expression of the form $pt_1\ldots t_n$ where p is an n-place predicate symbol of signature $<\sigma_1,\ldots,\sigma_n>$ and $t_1,\ldots,t_n$ are terms of sorts $\sigma_1,\ldots,\sigma_n$.

DLP also defines an expression called *atomic program*. This is not in a first-order language. If A is a procedure symbol of signature $<\sigma_1,\ldots,\sigma_n>\to<\tau_1,\ldots,\tau_m>$, $t_1,\ldots,t_n$ are terms of sorts $\sigma_1,\ldots,\sigma_n$, and $v_1,\ldots,v_n$ are individual variable symbols of sorts $\tau_1,\ldots,\tau_m$ then $A(v_1\ldots,v_m; t_1,\ldots,t_n)$ is an atomic program.

[11]The decision to call an operation that returns one or zero arguments a function or a procedure appears arbitrary at this point. The difference becomes clear when structures for DLP are discussed later in the section.

Formulas and programs are expressions defined by a simultaneous induction. Let G and H range over formulas and α and β range over programs. A *formula* is one of the following:

1. atomic formula
2. G & H
3. $G \vee H$
4. $\neg G$
5. $G \Rightarrow H$
6. $[\alpha]G$

A *program* is one of the following:

1. atomic program
2. $\alpha;\beta$
3. $\alpha \cup \beta$
4. α^*
5. G?

The DLP specification of the data type bounded stack of integers of size 100 as presented in [wand 82a] is:

1. nonlogical symbols
   a. sort symbols:

      stk
      int
      bool

   b. predicate symbols:

      $<$: $\langle int,int \rangle \rightarrow bool$
      $>$: $\langle int,int \rangle \rightarrow bool$
      $=$: $\langle int,int \rangle \rightarrow bool$
      $=_{stk}$: $\langle stk,stk \rangle \rightarrow bool$

   c. function symbols:

      LENGTH : $\langle stk \rangle \rightarrow int$

   d. constant symbols:

      false, **true** : bool
      1,2,3,... : int

   e. individual variable symbols:

      s_o : stk
      s : stk

n : int
t : stk

f. procedure symbols:

INIT: <>→<stk>
PUSH: <stk,int>→<stk>
POP: <stk>→<stk>

2. formulas

a. ∀s ([INIT(s;)] LENGTH(s)=0)

b. ∀s $\forall s_o$ ∀n (LENGTH(s_o)<100⇒[PUSH(s; n,s_o); POP(s; s)] $s =_{stk} s_o$

c. ∀s ∀t (LENGTH(s)=0⇒[POP(t; s)]**false**)

d. ∀s ∀t (LENGTH(s)>0⇒<POP(t; s)>**true**), where <α>G abbreviates ¬[α]¬G

For procedures, arguments to the left of the semi-colon are output parameters and those to the right are input parameters. A formula of the form [α]**false** asserts that **false** holds in any final state reached by the program α which is only possible if α never reaches a final state (i.e., α never halts on any input). A formula of the form <α>**true** asserts α halts on all inputs.

The nonlogical symbols and the set of formulas above comprise the *specification* or *theory* of bounded stacks of integers of size 100. Other abstract data types (e.g., arrays, lists) can be specified in the language DLP by specifying another set of nonlogical symbols and formulas. Another specification language can be defined by specifying the symbols and the syntax of the formulas in the language.

The reader may have noted that the three specifications of a bounded stack of integers of size 100 that were presented in this section do not define the same data type because the specifications differ in their treatment of error conditions. This can be attributed to differences in the specification languages.

B.3. Abstract Data Type Implementation

The stack example presented above has served to motivate and demonstrate the method for specifying abstract data types. Abstract data types are specified as theories. The *implementation* of an abstract data type is defined as an interpretation of the language of the theory for the abstract data type into another theory's language. This definition of implementation is based on an extension of interpretation between theories from first-order-logic (described in Appendix A) to DLP. The extension as described in [wand 82a] allows interpretations of procedure symbols, sorts, tuples of sorts, and equality symbols in addition to the nonlogical symbols in first-order logic. The extension requires that free variables in the interpreted programs and formulas be restricted to those values that are "legal" implementations of the variables' sort.

If L_1 and L_2 are DLP languages of theories T_1 and T_2, respectively, then an interpretation I of L_1 in L_2 is an assignment of phrases of L_2 to each nonlogical symbol of L_1 as follows:

1. **to each sort symbol** σ of L_1, a sort symbol σ^I of L_2 and a formula $\lambda x. is\text{-}\sigma(x)$ of signature σ^I; $I(\sigma)=\sigma^I$
2. **to each function symbol** f: $<\sigma_1,\ldots,\sigma_n>\rightarrow\tau$ of L_1, a function symbol f^I: $<\sigma_1^I,\ldots,\sigma_n^I>\rightarrow\tau^I$ of L_2; $I(f) = f^I$
3. **to each predicate symbol** p: $<\sigma_1,\ldots,\sigma_n>$ of L_1, a formula $p^I[z_1,\ldots,z_n]$ with signature $<\sigma_1^I,\ldots,\sigma_n^I>$ of L_2; $I(p) = p^I[z_1,\ldots, z_n]$
4. **to each individual variable symbol** v of L_1 with signature σ, an individual variable symbol v^I in L_2 with signature σ^I; $I(v) = v^I$
5. **for each procedure symbol** A: $<\sigma_1,\ldots,\sigma_n>\rightarrow<\tau_1,\ldots,\tau_n>$ a program $A^I[y_1,\ldots,y_m; z_1,\ldots,z_n]$ of L_2 with signature $<\sigma_1^I,\ldots,\sigma_n^I>\rightarrow<\tau_1^I,\ldots,\tau_n^I>$; $I(A) = A^I\ [y_1,\ldots,y_n;\ z_1,\ldots,z_n]$. In particular, $I(ASSIGN_\sigma) = (y_1:=z_1)$ and $I(FORALL_\sigma) = FORALL_{I(\sigma)}(y_1)$; $is\text{-}\sigma(y_1)$. Furthermore, no variable of the form v^I may appear in $A^I\ [y_1,\ldots,y_m;\ z_1,\ldots,z_n]$.

The arguments and results of interpretation I can be summarized as follows:

I: sort symbol→sort symbol

I: function symbol→ function symbol

I: predicate symbol→formula

I: procedure symbol→program

I: individual variable symbol→individual variable symbol

For DLP, a variable is *bound* if it is guaranteed to be set (assigned a value). This can only occur if it is an "output" parameter of a procedure (i.e., $\{v_1,\ldots,v_n\}$ are bound in procedure $A(v_1,\ldots,v_n;\ t_1,\ldots,t_n)$). If a variable is not bound, it is *free*.

Let G and H range over formulas and α and β over programs. Let preamble_G be the formula $(\text{is-}\sigma(x_1{}^I)\ \&\ \ldots\ \&\ \text{is-}\sigma_n(x_n{}^I))$ where $x_1,\ldots,x_n$ are the free variables of G and the free variables have sorts $\sigma_1,\ldots,\sigma_n$, respectively. The interpretation of G is $(\text{preamble}_G \Rightarrow I(G))$ where the interpretation between languages is extended as follows:

1. for a term $ft_1\ldots t_n$, $I(ft_1\ldots t_n) = f^I\,(I(t_1),\ldots,I(t_n))$
2. for an atomic formula $pt_1\ldots t_n$, $I(pt_1\ldots t_n) = [z_1 := I(t_1);\ \ldots\ ;\ z_n := I(t_n)]\ p^I(z_1,\ldots,z_n)$
3. for an atomic program $A(v_1,\ldots,v_n;\ t_1,\ldots,t_m)$, $I(A(v_1,\ldots,v_n;\ t_1,\ldots,t_m)) = (z_1 := I(t_1);\ \ldots\ ;\ z_m := I(t_m);\ A^I(y_1,\ldots,y_n;\ z_1,\ldots,z_m)\ v_1^I := y_1;\ \ldots\ ;\ v_n^I := y_n)$
4. for formulas,
   a. $I(G\ \&\ H) = (I(G)\ \&\ I(H))$
   b. $I(G \vee H) = (I(G) \vee I(H))$
   c. $I(\neg G) = (\neg I(G))$
   d. $I(G \Rightarrow H) = (I(G) \Rightarrow I(H))$
   e. $I([\alpha]G) = ([I(\alpha)]I(G))$

B.4. Abstract Data Type Semantics

A structure for a first-order language is a function that assigns functions and predicates to the function symbols and predicate symbols of the language, respectively. A structure for DLP is also an assignment of "meanings" or semantics to the set of non-logical symbols and the meanings are extended to apply to formulas and programs. A *structure* M is given as a function on each language symbol as follows:

1. **sort symbol**: for each sort symbol σ, $M(\sigma)=U_\sigma$ where U_σ is a nonempty set. U_σ is called the *carrier* of sort σ. U denotes the union of the sets U_σ as σ ranges over the sort symbols.
2. **function symbol**: for each function symbol f: $<\sigma_1,\ldots,\sigma_n> \rightarrow \sigma$, M assigns a function f^M: $U_{\sigma_1} \times \ldots \times U_{\sigma_n} \rightarrow U_\sigma$.
3. **predicate symbol**: for each predicate symbol p: $<\sigma_1,\ldots,\sigma_n>$, M assigns a predicate p^M on $U_\sigma \times \ldots \times U_{\sigma_n}$, such that for the distinguished predicate symbol$=_\sigma$, M assigns $=^M_\sigma$, the equality predicate on $U_\sigma \times U_\sigma$.
4. **procedure symbol**: for each procedure symbol $A:<\sigma_1,\ldots,\sigma_n> \rightarrow <\tau_1,\ldots,\tau_m>$, M assigns a predicate p^M_A on $U_{\sigma_1} \times \ldots \times U_{\sigma_n} \times U_{\tau_1} \times \ldots \times U_{\tau_m}$.

The arguments and results of the structure M, a function on the language symbols, can be summarized in the following way:

M: sort symbol → carrier

M: function symbol → function

M: predicate symbol → predicate

M: procedure symbol → predicate

A *state* ρ is a function from the set of individual variable symbols to U (i.e., ρ: variables →U).[12] A state is *sort-preserving* in the sense that if υ is an individual variable symbol of sort σ, then $\rho(\upsilon) \in U_\sigma$. M is extended to terms by mapping a term

[12]This is analogous to the function **s**, called the environment, for first order logic described in the Appendix A. **s** is only concerned with variables of a single sort.

to a function where the function maps a state to a value in one of the carriers (i.e., M: terms→states→U).[13] Specifically,

1. if x is an individual variable symbol then $M[\![x]\!](\rho) = \rho(x)$.
2. if $t_1,\ldots,t_n$ are terms of sorts $\sigma_1,\ldots\sigma_n$ and f is an n-place function symbol of signature $<\sigma_1,\ldots,\sigma_n>\rightarrow\sigma$, then $M[\![ft_1\ldots t_n]\!](\rho) = f^M(M[\![t_1]\!](\rho),\ldots,M[\![t_n]\!](\rho))$.

Now consider the extension of M to formulas and programs. M is extended to formulas by mapping formulas to functions that map states to boolean values (i.e., M: formulas → states → bool). M is extended to programs by mapping programs to functions that map a state to a set of states (i.e., M: programs→states→ 2^{states}). Since formulas and programs are defined by mutual recursion, their meanings are also defined by mutual recursion as follows:

1. if $pt_1\ldots t_n$ is an atomic formula then
$M[\![pt_1\ldots t_n]\!](\rho) = p^M(M[\![t_1]\!](\rho),\ldots, M[\![t_n]\!](\rho))$
2. if $A(v_1,\ldots, v_n; t_1,\ldots, t_m)$ is an atomic program then
$M[\![A(v_1,\ldots, v_n; t_1,\ldots, t_m)]\!](\rho) =$
$\{\rho' \mid p_A^M(\rho'(v_1),\ldots,\rho'(v_n), M[\![t_1]\!](\rho),\ldots, M[\![t_m]\!](\rho))$
$\& (\forall w)(w \notin \{v_1,\ldots, v_n\} \Rightarrow \rho(w) = \rho'(w))\}$
3. $M[\![G \& H]\!](\rho) = M[\![G]\!](\rho) \& M[\![H]\!](\rho)$
4. $M[\![G \vee H]\!](\rho) = M[\![G]\!](\rho) \vee M[\![H]\!](\rho)$
5. $M[\![\neg G]\!](\rho) = \neg M[\![G]\!](\rho)$
6. $M[\![G \Rightarrow H]\!](\rho) = M[\![\neg G]\!](\rho) \vee M[\![H]\!](\rho)$
7. $M[\![[\alpha]G]\!](\rho) = (\forall\rho')(\rho'\in M[\![\alpha]\!](\rho)\Rightarrow M[\![G]\!](\rho'))$
8. $M[\![\alpha; \beta]\!](\rho) = \{\rho'' \mid (\exists\rho')(\rho'\in M[\![\alpha]\!](\rho) \text{ and } \rho''\in M[\![\beta]\!](\rho'))\}$
9. $M[\![\alpha \cup \beta]\!](\rho) = M[\![\alpha]\!](\rho) \cup M[\![\beta]\!](\rho)$
10. $M[\![\alpha^*]\!](\rho)$ = the reflexive, transitive closure of $M[\![\alpha]\!](\rho)$
11. $M[\![G?]\!](\rho) = \{\rho \mid M[\![G]\!](\rho)\}$

M is a *model* of the theory if M satisfies every formula of the theory with every

[13]The notation in this dissertation differs from [wand 82a].

state ρ. A model for the specification of a bounded stack of integers of size 100 is the following:

1. **carriers (for each sort symbol):**
   M(int) = ω, the set of nonnegative integers
   M(stk) = ω^*, all finite strings of ω
   M(bool) = {**true**, **false**}
2. **predicates (for each predicate symbol):**
   M(<) = $<^M$, less than
   M(>) = $>^M$, greater than
   M(=) = $=^M$, equality of integer arguments
   M($=_{stk}$) = $=^M_{stk}$, equality of stack arguments
3. **functions (for each function symbol):**
   M(LENGTH)(x) = $|x|$, the number of integers in the finite string of integers, **x**
4. **predicates (for each procedure symbol):**
   M(INIT) = $\lambda s.\ s = \Lambda$
   M(PUSH) = $\lambda sns'.\ s' = n_1 \ldots n_k \Rightarrow s = nn_1 \ldots n_k$
   M(POP) = $\lambda ss'.\ (\exists k)\ k \geq 1\ \&\ s' = n_1 \ldots n_k\ \&\ s = n_2 \ldots n_k$

B.5. An Implementation is not a Model

In choosing a model for stacks an "abstract representation" was selected for each object type (sort symbol). For example, a stack is represented by a string of integers. This model is similar to the first stack specification presented in this chapter. In the model, each object has a unique abstract representation. The model can be considered an "implementation" of the specification, but in a typical implementation, there may be many representations for each object in the data type. These representations are "equivalent" if they represent the same object of an abstract data type.

For example, consider again the implementation of a stack, but this time the bounded stack of integers of size 100 is implemented (represented) as a pair of data types: an array of integers with dimension 1 to 100, and an integer (used as a pointer to the array). Let I be this particular implementation of bounded stacks. In [wand 82a] I is an interpretation of the theory of stacks into the theory of array-integer pairs.

In order to define the implementation I, the theory of array-integer pairs must be specified, and the interpretation of the language of stacks into the language of array-integer pairs must be specified. First, the theory of array-integer pairs is defined as:

1. sort symbols:

   arr
   int
   bool
   rec

2. predicate symbols:

   =: <int, int>→bool
   $=_{arr:}$<arr, arr>→bool
   $=_{rec:}$<rec, rec>→bool

3. function symbols:

   pair: <arr, int>→rec
   pr1: <rec>→arr
   pr2: <rec>→int

4. constant symbols:

   false,**true**: bool
   1, 2, 3, ...: int

5. individual variable symbols:

   a, a_o, a_1: arr
   i, j, n: int
   r, r_o, r_1, r_2, r′: rec
   r^L, r_o^L: arr
   r^R,r_o^R: int

6. procedure symbols:

   INITARRAY: <int>→<arr>
   FETCH: <arr,int>→<int>
   UPDATE: <arr,int,int>→<arr>

7. formulas[14]:

   a. $(\forall r)$ pair(pr1 (r),pr2 (r)) = r

[14]The set of formulas given here is not complete. A few formulas are presented to show how some properties of array-interger pairs might be specified. The specification of the assignment procedure with array arguments would require a lengthy discussion of substitution.

b. $(\forall r^L)(\forall r^R)$ pr1 (pair $(r^L, r^R) = r^L$

c. $(\forall r^L)(\forall r^R)$ pr2 (pair $(r^L, r^R) = r^R$

d. $(\forall n)(\forall a)(\forall i)(\forall m)(0 \leq i \leq m \Rightarrow$ [INITARRAY(a; m); FETCH(n; a,i)](n = 0))

e. $(\forall n)(\forall a_o)(\forall i)(\forall a)$[FETCH(n; a_o,i); UPDATE(a; a_o,i,n)](a = a_o)

Define the interpretation I of the language of stacks into the theory of array-integer pairs as follows:

1. **assign a sort symbol to each sort symbol**:

   I(stk)=rec
   I(int)=int
   I(bool)=bool

2. **assign a formula to each sort symbol**:

   is-stk = $\lambda r.$ pr2(r) ≥ 0
   is-int = $\lambda i.$ **true**
   is-bool = $\lambda b.$ **true**

   (N.B., $T_{arr\text{-}int} \models \exists r$ (pr2(r) ≥ 0))

3. **assign a formula to each quantifier symbol** σ:

   $I(\forall_\sigma) = \lambda x.\ \forall_{I(\sigma)} x$ (is-σ(x))

4. **assign a function symbol to each function symbol**:

   I(LENGTH) = pr2

5. **assign a formula to each predicate symbol**:

   $I(=_{stk}) = \lambda r_1 r_2.(\text{pr2}(r_1)=\text{pr2}(r_2))\ \&\ (\forall_{int} i)(1 \leq i \leq \text{pr2}(r_1) \Rightarrow$
   [FETCH (n_1; pr1(r_1), i); FETCH(n_2; pr1(r_2),i)]($n_1 = n_2$)))

   I(op) = $\lambda ij.$ (i op j), where op $\in$ {=,<,>}

6. **assign a variable symbol to each variable symbol**:

   I(s) = r
   I(s_o) = r_o
   I(n) = n
   I(t) = r_1

7. **assign a program to each procedure symbol**:

   I(INIT) = $\lambda r.$ [INITARRAY(a;100); ASSIGN (r; pair (a,0))]
   I(POP) = $\lambda r\ r'.$ [pr2(r') > 0?; ASSIGN (r; pair (pr1(r'), pr2(r')-1))]
   I(PUSH) = $\lambda r\ n\ r'.$ [pr2(r') < 100?; ASSIGN_{arr}(x; pr1(r'));
   UPDATE (x; x, pr2(r') + 1, n);
   ASSIGN_{rec}(r; pair (x, pr2(r') + 1))]
   $I(\text{ASSIGN}_\sigma) = \text{ASSIGN}_{I(\sigma)}$

I is not a model for the theory of bounded stacks of integers because equality is interpreted as an equivalence relation, not as equality in the theory of array-int pairs. In particular, consider the implementation of $=_{stk}$. The second formula in the theory of stacks is:

$(\forall s{:}stk)(\forall s_o{:}stk)(\forall n{:}int)(LENGTH(s_o) < 100 \Rightarrow [PUSH(s;\ n,s_o);\ POP(s;\ s)]\ (s =_{stk} s_o))$ (*)

If equality of stacks, $=_{stk}$, was interpreted as equality of records, $=_{rec}$, formula (*) would be false in the implementation because the interpretation of the formula would be:

$(\forall r{:}\ rec)(\forall r_o{:}\ rec)(\forall n{:}\ int)(pr2(r) \geq 0\ \&\ pr2(r_o) \geq 0 \Rightarrow (pr2(r_o) < 100 \Rightarrow$
$[[pr2(r_o) < 100\ ?;\ ASSIGN_{arr}(x;\ pr1(r_o));\ UPDATE\ (x;\ x, pr2(r_o)+1, n);$
$ASSIGN_{rec}\ (r;\ pair\ (x,\ pr2(r_o) + 1))]$
$[pr2(r) > 0\ ?;\ ASSIGN_{rec}\ (r;\ pair\ (pr1(r),\ pr2(r)-1))\ ]\ ]$
$(r =_{rec} r_o)))$

This can be easily demonstrated by considering an example (an instance of the translated formula). Let s_o be the empty stack created by INIT. After executing [PUSH(s; 2, s_o); POP(s; s)] in the implementation the value of the implementation of s, r, is <(2,0,0,0,...),0>, but the value of the implementation of s_o, r_o, is <(0,0,0,0,...), 0>. So $r \neq_{rec} r_o$ and the interpretation of the second formula is false. Thus, equality of stacks should not be interpreted as equality in the implementation because there may be many representations for the same stack. However, in the correct implementation I described above, $=_{stk}$ was interpreted as the formula $(pr2(r) = pr2(r_o))\ \&\ (\forall i{:}int)\ (1 \leq i \leq pr2(r) \Rightarrow [FETCH(n;\ pr1(r), i)\ ;\ FETCH(n_o;\ pr1(r_o), i)]\ (n = n_o))$. With this interpretation of $=_{stk}$ as an equivalence relation the second formula is true in the implementation (notice that <(2,0,0,0,...),0> and <(0,0,0,0,...),0> are equivalent with this definition).

B.6. An Implementation is not a Homomorphism

Let α and β be structures for a language. A *homomorphism* h of α into β is a function h: $|\alpha| \rightarrow |\beta|$ such that

1. for each n-place predicate symbol P and each n-tuple $<a_1,...a_n>$ of elements of $|\alpha|$, $<a_1,...,a_n> \in P^\alpha$ iff $<h(a_1),...,h(a_n)> \in P^\beta$
2. for each n-place function symbol f and each n-tuple, $h(f^\alpha(a_1, ..., a_n)) = f^\beta(h(a_1), ..., h(a_n))$

These two conditions are usually stated as h *preserves* the relations and functions.

Consider a first-order language L with variables $x_1,..., x_k$ $(k \geq 1)$, n-place function symbols $f_1^n,..., f_l^n$ (n, $l \geq 1$), n-place predicate symbols $p_1^n,..., p_m^n$ (n, $m \geq 1$), and constant symbols $c_1,...,c_p$ $(p \geq 1)$. A *Herbrand Universe* for L is constructed as follows:

1. $\{x_1,...,x_k,c_1,...c_p,f_1^n,...,f_l^n\}$ are elements of the Herbrand Universe. Call this set H.
2. for $t_1,...,t_n \in H$, $f_i^n(t_1,...,t_n) \in H$ where n, $i \geq 1$

In other words a Herbrand Universe is composed of the symbols and terms of the language. The *Herbrand Base* for L is the set of formulas obtained when variables in the formulas of L are replaced by elements of H.

Another definition (other than the one given in Appendix A) of a structure for L is a mapping from the Herbrand Base to the set of boolean values, {true, false}. We can also define a structure for L as the Herbrand Universe. In this way, the "meaning" of each language element is the string of symbols denoting the language element. Call this structure defined as the Herbrand Universe S. There is a unique homomorphism from S to any other structure of L.[15]

---

[15]In algebra, S is called the word algebra or initial algebra, denoted T_Σ. An implementation is often defined as a homomorphism from S to another structure (algebra).

The implementation I is not a homomorphism from S because in an interpreted formula, quantification must be restricted to values of the variables in the implementation language which satisfy the formula of their sort. For example, the interpretation of a formula may not equal the interpretation of the predicate symbol applied to the interpretation of the arguments (i.e., $I(p(a_1,\dots,a_n)) \neq I(p)(I(a_1),\dots I(a_n)))$. Rather, if $a_1,\dots,a_n$ are variables of sorts $\sigma_1,\dots,\sigma_n$, respectively, then for implementation I, $I(p(a_1,\dots,a_n)) = \text{is-}\sigma_1(a_1)\&\dots\&\ \text{is-}\sigma_n(a_n) \Rightarrow I(p)\ I(a_1),\dots,I(a_n))$.

Consider the interpretation of formula (*) in the stack example. The quantification of s and s_o over stk is translated to the quantification of r and r_o over rec provided r and r_o satisfy the formula is-stk (i.e., $pr2(r) \geq 0$ & $pr2(r_o) \geq 0$). Again, it is easy to see the necessity of restricting r and r_o by considering an instance of r_o that does not satisfy is-stk. If r_o = <(1,0,...,0),-1> we have $\neg$ is-stk (r_o) and PUSH (s; 2, s_o) results in r = <(1,2,0,...,0),0> where r and r_o implement s and s_o, respectively. If this procedure is followed with the implementation of procedure POP(s; s) r does not change because pr2(r) = 0. Thus, the implementation of $s =_{stk} s_o$ does not hold after the implementation of [PUSH(s; 2,s_o); POP(s; s)]. So, formula (*) does not hold for all variables of type rec, but only those that "legally" represent variables of type stk.

B.7. An Abstract Data Type may be Implemented by Several Abstract Data Types

In the interpretation of a DLP language, predicate symbols were interpreted as formulas, procedure symbols were interpreted as programs, and sort symbols were interpreted as sort symbols. However, upon closer examination of the stack example it can be seen that while the interpretation of sort stk is the sort rec, rec is actually composed from two other sorts, arr and int. In other words, the theory of array-integer pairs is composed from the theory of arrays, the theory of integers, and function symbols and axioms that specify how to create objects of composite sorts and select components of these objects. The theory of array-integer pairs is called an *extension* of the theory of integers and arrays. The extension does not add information about the theory, but rather, adds definitions for convenience.

We will formally define an extension of theory T in language L to a theory T′ in language L′ below. In the stack example, T is the theory of integers and arrays and T′ is the theory of array-integer pairs.

Let σ_1 and σ_2 be sort symbols in L. If we require a composite sort constructed from the tuple of sorts $<\sigma_1, \sigma_2>$ then we modify L in the following way and call it L′.[16] Call the new sort symbol created from the tuple σ. Add to L the new sort symbol σ along with a countably infinite set of variables of sort σ, and function symbols pr1: $\sigma \to \sigma_1$, pr2: $\sigma \to \sigma_2$, and pair: $<\sigma_1, \sigma_2> \to \sigma$. For each variable x of sort σ, designate two variables x^L and x^R of sorts σ_1 and σ_2, respectively. Delete any existing variables of the form x^L and x^R in L.

The theory T′ is obtained by adding the following axioms to T:

1. pair (pr1(x), pr2(x)) = x
2. pr1(pair(x, y)) = x
3. pr2(pair(x, y)) = y

Intuitively, we desire the "untupled" version of any formula that is true in L′ to be true in L. That is, T′ does not contain any more information than T, but merely defines some useful abbreviations. The "untupled" version of a formula is made more precise below by defining a translation of formulas of L′ to formulas of L.

[16]This discussion can be generalized for tuples with any number of elements.

If t' is a term of L' of sort other than the new sort symbol σ, then a translation R from terms of L' to terms of L is defined as follows:

1. if t' is pr1(x), then $R(t') = x^L$
2. if t' is pr2(x), then $R(t') = x^R$
3. if t' is $pr1(pair(t_1,t_2))$, then $R(t') = R(t_1)$
4. if t' is $pr2(pair(t_1,t_2))$, then $R(t') = R(t_2)$
5. if t' is a variable, then $R(t') = t'$
6. if $t' = ft_1..t_n$ and $f \notin \{pr1, pr2, pair\}$, then $R(t') = f\, R(t_1)...R(t_n)$

The translation R is extended to programs and formulas by doing the following substitutions:

1. $t_1 =_\sigma t_2$ is replaced by $(R(pr1(t_1)) =_{\sigma_1} R(pr1(t_2))\ \&\ R(pr2(t_1)) =_{\sigma_2} R(pr2(t_2)))$
2. $(\forall_\sigma x)$ is replaced by $\forall_{\sigma_1} x^L; \forall_{\sigma_2} x^R$
3. x:=t where x is of sort σ is replaced by z_1:=R(pr1(t)); z_2:=R(pr2(t)); x^L:=z_1; x^R:=z_2 and z_1 and z_2 are variables which appear nowhere else in the formula

Theory T' is an *extension by definitions* of T iff T' is obtained from T by repeatedly adding new sorts in the manner described above.

Theorem 1: If T' is an extension by definitions of T, G is a formula in the language L' and R defines the translation from formulas in L' to formulas of L, then $T' \models G$ iff $T \models R(G)$

The proof of this theorem is in [wand 82a].

This theorem means an implementation can be expressed in terms of several abstract data types. Extending a theory by adding tuples of sorts does not make the theory more powerful as long as the new symbols are well defined (i.e., are function symbols).

B.8. Correct Implementations

How do we know I is a correct implementation of bounded stacks? Intuitively, for the stack example any property of bounded stacks should be preserved in the implementation. Stated more formally, if a formula is true in the theory of bounded stacks then its interpretation should be true in the theory of array-integer pairs.

To define the conditions of a correct implementation, we introduce the concept of interpretation of one theory into another theory. If T_1 is a theory in language L_1, and T_2 is a theory in language L_2, then an *interpretation of T_1 in T_2* is an interpretation I of L_1 in L_2 such that the following formulas are logical consequences of T_2:

1. $\exists x(\text{is-}\sigma(x))$ for each sort σ of L_2[17]
2. $\text{is-}\sigma_1(x_1)\ \&\ldots\&\ \text{is-}\sigma_n(x_n) \Rightarrow \text{is-}\sigma(f^I\ x_1 \ldots x_n)$ for each function symbol f: $\langle\sigma_1,\ldots,\sigma_n\rangle \rightarrow \sigma$ in L_1
3. $\text{is-}\tau_1(z_1)\ \&\ldots\&\ \text{is-}\tau_n(z_n) \Rightarrow [A^I]\ \text{is-}\sigma_i(y_i)$ for each procedure symbol A: $\langle\tau_1,\ldots,\tau_n\rangle \rightarrow \langle\sigma_1,\ldots,\sigma_n\rangle$ and interpretation $A^I[y_1,\ldots,y_n; z_1,\ldots,z_n]$, and $1 \le i \le n$
4. $I(x =_\sigma x)$ for each sort σ of L_1
5. $I(x_1 = y_1\ \&\ldots\&\ x_n = y_n \Rightarrow (fx_1 \ldots x_n = fy_1 \ldots y_n))$[18]
6. $I(x_1 = y_1\ \&\ldots\&\ x_n = y_n \Rightarrow (px_1 \ldots x_n \Rightarrow py_1 \ldots y_n))$
7. I(G) for each axiom G of T_1

Conditions 1 and 5 correspond to conditions for first-order theories. Conditions 2 and 3 are required because we have introduced sorts into the language. They state that if the input data satisfy the formula (invariant) of their sort, then the output of the interpreted function or procedure satisfies the formula (invariant) of its sort.

[17]This corresponds to the condition for first-order theories that $T_2 \models \exists v_1 \pi_\forall$ where $\pi_\forall$ is the formula assigned to ∀ by the interpretation.

[18]This corresponds to the condition for first-order theories that $T_2 \models \forall v_1 \ldots \forall v_n\ (\pi_\forall(v_1) \Rightarrow \ldots \Rightarrow \pi_\forall(v_n) \Rightarrow \exists x\ (\pi_\forall(x)\ \&\ \forall v_{n+1}\ (\pi_f\ v_1,\ldots,v_n = v_{n+1} \Rightarrow v_{n+1} = x)))$ where the interpretation assigns the formula π_f of L_2 to function symbol f. Though the formulas are in different in form, they state the same condition: the interpretation preserves functions.

Items 4, 5, and 6 are necessary because equality may be interpreted as an equivalence relation. They state the interpretation of equality is a reflexive relation and is preserved by the interpretation of terms and predicates. Item 5 states that the interpretation of a function symbol is a function. Item 7 states that the translation of the axioms of T_1 are logical consequences of T_2.

A correct *implementation* of a theory T_1 in a theory T_2 is an interpretation I of T_1 in T_2 where T_2 may be an extension by definitions of a theory. The main theorem proved in [wand 82a] is

Theorem 2: (The Implementation Theorem). Let I be a correct implementation of T_1 in T_2.

1. If A is any L_2-structure, then there is an L_1-structure A' such that for any closed formula G of L_1, $A' \models G$ iff $A \models I(G)$.[19]
2. For any formula G of L_1, if $T_1 \models G$, then $T_2 \models I(G)$.[20]

B.9. Correctness Proofs

In this chapter an example of a correctness proof is presented. However, as described below the stack example is not used. Let T_{stk} be the theory of bounded stacks and $T_{arr\text{-}int}$ be the theory of array-integer pairs. If I is a correct implementation of T_{stk} into $T_{arr\text{-}int}$, then if α is a valid sentence in T_{stk} then $I(\alpha)$ is a valid sentence in $T_{arr\text{-}int}$. If T_{stk} and $T_{arr\text{-}int}$ are sound and complete then to prove the last condition above (condition #7) it is sufficient to show that the interpretation of the axioms and rules of T_{stk} are deducible in $T_{arr\text{-}int}$. If T_{stk} and $T_{arr\text{-}int}$ are not complete we prove a more restricted result: the interpretation of any formula deducible in T_{stk} is deducible in $T_{arr\text{-}int}$. If T_{stk} and $T_{arr\text{-}int}$ are sound and the

[19] This corresponds to Lemma 1 in Appendix A for first-order theories.

[20] This corresponds to interpretation of one theory into another for first-order theories.

formulas in T_{stk} are restricted to those deducible in T_{stk} then this proof will be sufficient to show that the translation of the valid T_{stk} sentences are valid in $T_{arr\cdot int}$.

A specification of the stack data type, a partial specification of the array-integer pair data type, and the implementation of stacks using array-integer pairs were discussed above. In a complete specification of array-integer pairs we would have axioms specifying the assignment procedures with array argument types. These axioms are rather complicated to specify and require a lengthy discussion of substitution. Consequently, we have chosen another example for the purpose of demonstrating the correctness proof technique. Consider the following simple implementation of a data type whose only operation is SWITCH. Let T_{spec} be the theory for the abstract data type that we want to implement and let T_{impl} be the theory in which T_{spec} is implemented in. T_{spec} is implemented in T_{impl}. Define T_{spec} as follows:

1. Language
   a. sort symbols:
      int
      bool
   b. predicate symbols:
      $=:\langle int,int\rangle$
   c. individual variable symbols:
      a,b,x,y,x_o,y_o: int
   d. procedure symbols:
      SWITCH: $\langle int,int\rangle\rightarrow\langle int,int\rangle$
2. Axioms
   a. $((x=x_o)\&(y=y_o))\Rightarrow[SWITCH(a,b;\ x,y)]((a=y_o)\&(b=x_o))$
   b. $((x=x_o)\&(y=y_o))\Rightarrow[SWITCH(x,y;\ x,y)]((x=y_o)\&(y=x_o))$

Define T_{impl} as follows:

1. Language
   a. sort symbols:
      int

b. individual variable symbols:

x,t: int
P,R,Q: formula
α,β: program

2. Axioms:[21]

a. $P[t/x] \Rightarrow [ASSIGN_{int}(x;\ t)]P$

3. Rules:

a. $\dfrac{P \Rightarrow [\alpha]R,\ R \Rightarrow [\beta]}{P \Rightarrow [\alpha;\ \beta]Q}$

Define IMP as an implementation of T_{spec} into T_{impl} as follows:

IMP(int)=int
IMP(SWITCH(a,b; x,y))=$[ASSIGN_{int}(t;x);\ ASSIGN_{int}(a;\ y);\ ASSIGN_{int}(b;t)]$

As part of the proof to show that IMP is a correct implementation we must show that the interpretation of both axioms in T_{spec} are deducible in T_{impl}. The interpretation of the first axiom in T_{spec} is

$((x=x_o)\&(y=y_o)) \Rightarrow [ASSIGN_{int}(t;\ x);\ ASSIGN_{int}(a;\ y);\ ASSIGN_{int}(b;\ t)]((a=y_o)\ \&\ (b=x_o))$

The proof of this in T_{impl} is:

(1) $((a=y_o)\&(t=x_o)) \Rightarrow [ASSIGN_{int}(b;\ t)]\ ((a\text{-}y_o)\ \&\ (b=x_o))$ (axiom)

(2) $((y=y_o)\ \&\ (t=x_o)) \Rightarrow [ASSIGN_{int}(a;\ y)]\ ((a=y_o)\ \&\ (t=x_o))$ (axiom)

(3) $((y=y_o)\&(t=x_o)) \Rightarrow [ASSIGN_{int}(a;\ y);\ ASSIGN_{int}(b;\ t)]((a=y_o)\&(b=x_o))$ ((1),(2), and rule)

(4) $((y=y_o)\&(x=x_o)) \Rightarrow [ASSIGN_{int}(t;\ x)]((y=y_o)\&(t=x_o))$ (axiom)

(5) $((x=x_o)\&(y=y_o)) \Rightarrow$
$[ASSIGN_{int}(t;\ x);\ ASSIGN_{int}(a;\ y);\ ASSIGN_{int}(b;\ t)]((a=y_o)\ \&\ (b=x_o))$ ((3),(4), and rule)

The interpretation of the second axiom is

$((x=x_o)\ \&\ (y=y_o) \Rightarrow [ASSIGN_{int}(t;\ x);\ ASSIGN_{int}(x;\ y);\ ASSIGN_{int}(y;\ t)]((x=y_o)\ \&\ (y=x_o)$

The proof of this in T_{impl} is similar to the proof of the first axiom above.[22] Suppose we had interpreted the procedure SWITCH as IMP(SWITCH(a,b; x,y))=$[ASSIGN_{int}(a;\ y);$

[21] $P[t/x]$ means substitute t for all free occurrences of x in P.

[22] These proofs use methods of Floyd, Hoare and Dijkstra.

$ASSIGN_{int}(b; x)$]. Then the interpretation of the first axiom would be true in T_{impl}, but the interpretation of the second axiom would be false. The second axiom asserts a property of side effects with the input variables (the values of input variables are altered in the procedure SWITCH), and the correct implementation uses a "temporary" variable t to preserve the desired property.

Appendix C

Implementation of DS-Tiny

Theory for Source Language, T_{source}

Language for Source Language, L_{source}

Language Elements	Defined Language	Defining Language
domains	id = {**I**, $\mathbf{I_1}$, $\mathbf{I_2}$...} exp com prog	num bool value = num ⊕ bool input = value* output = value* mem = id → [value ⊕ {**unbound**}] state = mem ⊗ input ⊗ output
function symbols	**0**: → exp	*E*: exp → (state → ((value ⊗ state) ⊕ {error}))
	1: → exp	*C*: com → (state → (state ⊕ {error}))
	true: → exp	*P*: prog → input → [output ⊕ {error}]
	false: → exp	***hd***: value* → value ⊕ {error}
	read: → exp	***tl***: value* → value*
	{**I**, $\mathbf{I_1}$, $\mathbf{I_2}$...}: → exp	_ • _: value ⊗ value* → value*
	not: exp → exp	_+_: num ⊗ num → num
	=: exp ⊗ exp → exp	
	+: exp ⊗ exp → exp	
	:=: id ⊗ exp → com	
	output: exp → com	
	if: exp ⊗ com ⊗ com → com	
	while: exp ⊗ com → com	
	;: com ⊗ com → com	
	begin: com → prog	

predicate symbols	null: value* → bool
individual variable symbols	v, v′, v_i: value, $1 \le i \le n$ m: mem i: input o: output C, C_1, C_2: com E, E_1, E_2: exp I, I′: id v*: value* s: state P: prog (m,i,o): state b: bool

Axioms for Source Language, A_{source}

(E1a) $\mathcal{E}[\![\,\mathbf{0}\,]\!](s) =_{((value \otimes state) \oplus \{error\})} \langle 0, s\rangle$

(E1b) $\mathcal{E}[\![\,\mathbf{1}\,]\!](s) =_{((value \otimes state) \oplus \{error\})} \langle 1, s\rangle$

(E2a) $\mathcal{E}[\![\,\mathbf{true}\,]\!](s) =_{((value \otimes state) \oplus \{error\})} \langle TRUE, s\rangle$

(E2b) $\mathcal{E}[\![\,\mathbf{false}\,]\!](s) =_{((value \otimes state) \oplus \{error\})} \langle FALSE, s\rangle$

(E3) $\mathcal{E}[\![\,\mathbf{read}\,]\!](\langle m, i, o\rangle) =_{((value \otimes state) \oplus \{error\})}$
$null(i) \to error, \langle \mathbf{\mathit{hd}}(i), \langle m, \mathbf{\mathit{tl}}(i), o\rangle\rangle$

(E4) $\mathcal{E}[\![\,I\,]\!](\langle m, i, o\rangle) =_{((value \otimes state) \oplus \{error\})}$
$m(I) = \mathbf{unbound} \to error, \langle m(I), \langle m, i, o\rangle\rangle$

(E5) $\mathcal{E}[\![\,\mathbf{not}\ E\,]\!](s) =_{((value \otimes state) \oplus \{error\})}$
$(\mathcal{E}[\![\,E\,]\!](s) = \langle v, s'\rangle) \to [\text{is-bool}(v) \to \langle \sim v, s'\rangle, error], error$

(E6) $\mathcal{E}[\![\,E_1 = E_2\,]\!](s) =_{((value \otimes state) \oplus \{error\})}$
$(\mathcal{E}[\![\,E_1\,]\!](s) = \langle v_1, s_1\rangle) \to$
$((\mathcal{E}[\![\,E_2\,]\!](s_1) = \langle v_2, s_2\rangle) \to \langle v_1 = v_2, s_2\rangle, error), error$

(E7) $\mathcal{E}[\![\,E_1 + E_2\,]\!](s) =_{((value \otimes state) \oplus \{error\})}$
$(\mathcal{E}[\![\,E_1\,]\!](s) = \langle v_1, s_1\rangle) \to ((\mathcal{E}[\![\,E_2\,]\!](s_1) = \langle v_2, s_2\rangle) \to$
$[\text{is-num}(v_1)\ \&\ \text{is-num}(v_2) \to \langle v_1 + v_2, s_2\rangle, error], error), error$

(C1) $\mathcal{C}[\![\,I := E\,]\!](s) =_{(state \oplus \{error\})}$

$(\boldsymbol{E}\llbracket E \rrbracket(s) = \langle v, \langle m, i, o\rangle\rangle) \rightarrow \langle m[v/I], i, o\rangle, \text{error}$

(C2) $\boldsymbol{C}\llbracket \textbf{output } E \rrbracket(s) =_{(state \oplus \{error\})}$
$(\boldsymbol{E}\llbracket E \rrbracket(s) = \langle v, \langle m, i, o\rangle\rangle) \rightarrow \langle m, i, o \bullet v\rangle, \text{error}$

(C3) $\boldsymbol{C}\llbracket \textbf{if } E\ C_1\ C_2 \rrbracket(s) =_{(state \oplus \{error\})}$
$(\boldsymbol{E}\llbracket E \rrbracket(s) = \langle v, s'\rangle) \rightarrow [\text{is-bool}(v) \rightarrow$
$(v \rightarrow \boldsymbol{C}\llbracket C_1 \rrbracket(s'),\ \boldsymbol{C}\llbracket C_2 \rrbracket(s')), \text{error}], \text{error}$

(C4) $\boldsymbol{C}\llbracket \textbf{while } E\ C \rrbracket(s) =_{(state \oplus \{error\})}$
$(\boldsymbol{E}\llbracket E \rrbracket(s) = \langle v, s'\rangle) \rightarrow [\text{is-bool}(v) \rightarrow (v \rightarrow ((\boldsymbol{C}\llbracket C \rrbracket(s') = s'') \rightarrow$
$\boldsymbol{C}\llbracket \textbf{while } E\ C \rrbracket(s''), \text{error}), s'), \text{error}], \text{error}$

(C5) $\boldsymbol{C}\llbracket C_1 ; C_2 \rrbracket(s) =_{(state \oplus \{error\})}$
$(\boldsymbol{C}\llbracket C_1 \rrbracket(s) = \text{error}) \rightarrow \text{error}, \boldsymbol{C}\llbracket C_2 \rrbracket(\boldsymbol{C}\llbracket C_1 \rrbracket(s))$

(P1) $\boldsymbol{P}\llbracket \textbf{begin } P \rrbracket(i) =_{output \oplus \{error\}} \lambda a.\ [a = \text{error} \rightarrow a,\ \boldsymbol{hd}(\boldsymbol{tl}(\boldsymbol{tl}(a)))]$
$(\boldsymbol{C}\llbracket P \rrbracket(m_o, i, \langle\rangle))$

where
$\forall I \in id.\ m_o(I) = \textbf{unbound}$
$\langle\rangle$ = initially empty output

(A1) $m[v/I](I') =_{value \oplus \{\textbf{unbound}\}} (I =_{id} I' \rightarrow v, m(I'))$

(A2a) $\boldsymbol{hd}(\langle\rangle) = \text{error}$

(A2b) $\boldsymbol{hd}(\langle v\rangle) = v$

(A2c) $\boldsymbol{hd}(\langle v\rangle \bullet v^*) = v$

(A2d) $\boldsymbol{tl}(\langle v\rangle \bullet v^*) = v^*$

(A2e) $\boldsymbol{hd}(v^*) \bullet \boldsymbol{tl}(v^*) = v^*$

Theory for Target Language, T_{target}

Language for Target Language, L_{target}

Language Elements	Defined Language	Defining Language
domains	id instr code = instr* ecode = instr* num bool pcode = instr*	value = num ⊕ bool stack = value* mem = id → [value ⊕ {**unbound**}] input = value* output = value* state = mem ⊗ input ⊗ output mstate = stack ⊗ state

function symbols	**start**: → instr	*MI*: instr → (mstate → (mstate ⊕ {error}))
	halt: → instr	*MC*: code → (mstate → (mstate ⊕ {error}))
	loadn: num → instr	*ME*: ecode → (mstate → (mstate ⊕ {error}))
	loadb: bool → instr	*MP*: pcode → (mstate → (mstate ⊕ {error}))
	read: → instr	*hd*: value* → value ⊕ {error}
	load: id → instr	*tl*: value* → value*
	not: → instr	*lg*: value* → num
	eq: → instr	_ • _: value ⊗ value* → value*
	add: → instr	_+_: num ⊗ num → num
	store: id → instr	
	output: → instr	
	cond: code ⊗ code → instr	
	loop: ecode ⊗ code → instr	
predicate symbols		null: value* → bool
		<: num ⊗ num → bool
individual variable symbols		stk: stack
		m: mem
		i: input
		o: output
		(stk,m,i,o): mstate
		ID: id
		P, Q: code
		T: ecode
		I: instr
		v: value

Axioms for Target Language, A_{target}

(I1) *MI* ⟦ [**loadn**, 0] ⟧ ((stk,m,i,o)) $=_{(\text{mstate} \oplus \{\text{error}\})}$ (<0>•stk,m,i,o)

(I2) *MI* ⟦ [**loadn**, 1] ⟧ ((stk,m,i,o)) $=_{(\text{mstate} \oplus \{\text{error}\})}$ (<1>•stk,m,i,o)

(I3) *MI* ⟦ [**loadb**, TRUE] ⟧ ((stk,m,i,o)) $=_{(\text{mstate} \oplus \{\text{error}\})}$ (<TRUE>•stk,m,i,o)

(I4) *MI* ⟦ [**loadb**, FALSE] ⟧ ((stk,m,i,o)) $=_{(\text{mstate} \oplus \{\text{error}\})}$ (<FALSE>•stk,m,i,o)

(I5) ***MI*** ⟦ [**read**] ⟧ ((stk,m,i,o)) $=_{(mstate \oplus \{error\})}$
[null(i) → error, (<***hd***(i)>•stk,m,***tl***(i),o)]

(I6) ***MI*** ⟦ [**load**,ID] ⟧ ((stk,m,i,o)) $=_{(mstate \oplus \{error\})}$
[m(ID) = **unbound** → error, (<m(ID)>•stk,m,i,o)]

(I7) ***MI*** ⟦ [**not**] ⟧ ((stk,m,i,o)) $=_{(mstate \oplus \{error\})}$
[***lg***(stk) < 1 → error,
(is-bool(***hd***(stk)) → (<~***hd***(stk)>•***tl***(stk),m,i,o), error)]

(I8) ***MI*** ⟦ [**eq**] ⟧ ((stk,m,i,o)) $=_{(mstate \oplus \{error\})}$
[***lg***(stk) < 2 → error,
(<***hd***(***tl***(stk)) = ***hd***(stk)>•***tl***(***tl***(stk)),m,i,o)]

(I9) ***MI*** ⟦ [**add**] ⟧ ((stk,m,i,o)) $=_{(mstate \oplus \{error\})}$
[***lg***(stk) < 2 → error,
(is-num(***hd***(stk)) & is-num(***hd***(***tl***(stk))) →
(<***hd***(***tl***(stk)) + ***hd***(stk)>•***tl***(***tl***(stk)),m,i,o),
error)]

(I10) ***MI*** ⟦ [**store**, ID] ⟧ ((stk,m,i,o)) $=_{(mstate \oplus \{error\})}$
[***lg***(stk) < 1 → error, (***tl***(stk),m[***hd***(stk)/ID],i,o)]

(I11) ***MI*** ⟦ [**output**] ⟧ ((stk,m,i,o)) $=_{(mstate \oplus \{error\})}$
[***lg***(stk) < 1 → error, (***tl***(stk),m,i,o•<***hd***(stk)>)]

(I12) ***MI*** ⟦ [**cond**, P, Q] ⟧ ((stk,m,i,o)) $=_{(mstate \oplus \{error\})}$
[***lg***(stk) < 1 → error,
(is-bool(***hd***(stk)) →
(***hd***(stk) → ***MC*** ⟦ P ⟧ ((***tl***(stk),m,i,o)), ***MC*** ⟦ Q ⟧ ((***tl***(stk),m,i,o))),
error)]

(I13) ***MI*** ⟦ [**loop**, T, P] ⟧ (stk,m,i,o) $=_{(mstate \oplus \{error\})}$
ME ⟦ T ⟧ (λ(stk,m,i,o). [is-bool(***hd***(stk)) →
(***hd***(stk) → (***MI*** ⟦ [**loop**, T, P] ⟧ (***MC*** ⟦ P ⟧ ((***tl***(stk),m,i,o)))), ((***tl***(stk),m,i,o))),
error])

(I14) ***MI*** ⟦ [**start**] ⟧ ((stk,m,i,o)) $=_{(mstate \oplus \{error\})}$ (<>,m_0,i,<>)

(I15) ***MI*** ⟦ [**halt**] ⟧ ((stk,m,i,o)) $=_{(mstate \oplus \{error\})}$ (stk,m,i,o)

(TC1) ***MC*** ⟦ <> ⟧ (s) $=_{(mstate \oplus \{error\})}$ s

(TC2) ***MC*** ⟦ <I>•P ⟧ (s) $=_{(mstate \oplus \{error\})}$
MI ⟦ I ⟧ s = error → error, (***MC*** ⟦ P ⟧ (***MI*** ⟦ I ⟧ s))

(TC3) ***MC*** ⟦ P•Q ⟧ (s) $=_{(mstate \oplus \{error\})}$

$\boldsymbol{MC}\ [\![\ P\ ]\!]\ s = error \rightarrow error,\ (\boldsymbol{MC}\ [\![\ Q\ ]\!]\ (\boldsymbol{MC}\ [\![\ P\ ]\!]\ s))$

(TE1) $\boldsymbol{ME}\ [\![\ <>\ ]\!]\ (s) =_{(mstate \oplus \{error\})} s$

(TE2) $\boldsymbol{ME}\ [\![\ <I>\bullet P\ ]\!]\ (s) =_{(mstate \oplus \{error\})}$
$\boldsymbol{MI}\ [\![\ I\ ]\!]\ s = error \rightarrow error,\ (\boldsymbol{ME}\ [\![\ P\ ]\!]\ (\boldsymbol{MI}\ [\![\ I\ ]\!]\ s))$

(TE3) $\boldsymbol{ME}\ [\![\ P\bullet Q\ ]\!]\ (s) =_{(mstate \oplus \{error\})}$
$\boldsymbol{ME}\ [\![\ P\ ]\!]\ s = error \rightarrow error,\ (\boldsymbol{ME}\ [\![\ Q\ ]\!]\ (\boldsymbol{ME}\ [\![\ P\ ]\!]\ s))$

(TP1) $\boldsymbol{MP}\ [\![\ <>\ ]\!]\ (s) =_{(mstate \oplus \{error\})} s$

(TP2) $\boldsymbol{MP}\ [\![\ <I>\bullet P\ ]\!]\ (s) =_{(mstate \oplus \{error\})}$
$\boldsymbol{MI}\ [\![\ I\ ]\!]\ s = error \rightarrow error,\ (\boldsymbol{MP}\ [\![\ P\ ]\!]\ (\ \boldsymbol{MI}\ [\![\ I\ ]\!] s))$

(TP3) $\boldsymbol{MP}\ [\![\ P\bullet Q\ ]\!]\ (s) =_{(mstate \oplus \{error\})}$
$\boldsymbol{MP}\ [\![\ P\ ]\!]\ s = error \rightarrow error,\ (\boldsymbol{MP}\ [\![\ Q\ ]\!]\ (\boldsymbol{MP}\ [\![\ P\ ]\!]\ s))$

(A1) $m[v/I](I') =_{value \oplus \{\mathbf{unbound}\}} (I =_{id} I' \rightarrow v,\ m(I'))$

(A2a) $\boldsymbol{hd}(<>) = error$

(A2b) $\boldsymbol{hd}(<v>) = v$

(A2c) $\boldsymbol{hd}(<v>\bullet v^*) = v$

(A2d) $\boldsymbol{tl}(<v>\bullet v^*) = v^*$

(A2e) $\boldsymbol{hd}(v^*)\bullet \boldsymbol{tl}(v^*) = v^*$

Interpretation

Language Elements	Defined Language	Defining Language
$\boldsymbol{I}$: domain → domain	$\boldsymbol{I}$(id) = id $\boldsymbol{I}$(exp) = ecode $\boldsymbol{I}$(com) = code $\boldsymbol{I}$(prog) = pcode	$\boldsymbol{I}$(num) = num $\boldsymbol{I}$(bool) = bool $\boldsymbol{I}$(value) = value $\boldsymbol{I}$(input) = mstate $\boldsymbol{I}$(output) = mstate $\boldsymbol{I}$(mem) = mem $\boldsymbol{I}$(state) = mstate $\boldsymbol{I}(state_o) = mstate_o$

I: function symbol → term	*I*(**0**) = [**loadn**, 0] *I*(**1**) = [**loadn**, 1] *I*(**true**) = [**loadb**, TRUE] *I*(**false**) = [**loadb**, FALSE] *I*(**read**) = [**read**] *I*($\mathbf{I_i}$) = [**load**, $\mathbf{I_i}$], i ≥ 1 *I*(**not**) = λE. (E) • [**not**] *I*(**=**) = λE_1E_2. (E_1) • (E_2) • [**eq**] *I*(**+**) = λE_1E_2. (E_1) • (E_2) • [**add**] *I*(**:=**) = λIE. (E) • [**store**, I] *I*(**output**) = λE. (E) • [**output**] *I*(**if**) = λEC_1C_2. (E) • [**cond**, C_1, C_2] *I*(**while**) = λEC. [**loop**, E, C] *I*(**;**) = λC_1C_2. (C_1) • (C_2) *I*(**begin**) = λC. [**start**] • (C) • [**halt**]	*I*(***E***) = λEs. ***H*** (E) (s) where ***H***(C)((stk, m, i, o)) equals ***ME***(C)((stk, m, i, o)) = error → error, <***hd***(pr1(***ME***(C)((stk, m, i, o)))), <***tl***(pr1(***ME***(C)((stk, m, i, o)))), pr2(***ME***(C)((stk, m, i, o))), pr3(***ME***(C)((stk, m, i, o))), pr4(***ME***(C)((stk, m, i, o)))>> *I*(***C***) = λCs. ***MC*** (C) (s) *I*(***P***) = λP. ***MP*** (P) s_o, where s_o = <<>,m_o, i,<>> *I*(***hd***) = λs. ***hd***(s) *I*(***tl***) = λs. ***tl***(s)
predicate symbol → predicate symbol		*I*($=_{value}$) = $=_{value}$ *I*($=_{id}$) = $=_{id}$ *I*(null) = null
individual variable symbol → term		*I*((m,i,o): state) = (stk,m,i,o): mstate *I*(s: state) = (stk,m,i,o): mstate *I*((m_o,i,<>): initial state) = (<>,m_o,i,<>): initial mstate *I*(i:input) = (stk,m,i,o): mstate *I*(***hd***(***tl***(***tl***(m, i, o)))): output) = (stk, m, i, o): mstate

new predicates

is-state: mstate → bool
is-input: mstate → bool

N.B., there are other predicates. All these predicates are trivially true.

Example Correctness Proof

Axiom (E1a)

Translate Axiom into L_{target}

E ⟦ **0** ⟧ (s) = <0, s>

(translate axiom using interpretation, ***I***)

λEk. [***H*** (E) (k)] (***I***(**0**)) (***I***(s)) = <***I***(0), ***I***(s)>

(simplify)

H ([**loadn**, **0**]) ((stk, m, i, o)) = <0, <stk, m, i, o>>

(simplify)

(***ME*** ([**loadn**, **0**]) ((stk, m, i, o)) = error → error,
<***hd*** (pr1 (***ME*** ([**loadn**,**0**]) ((stk, m, i, o)))),
<***tl***(pr1(***ME*** ([**loadn**,**0**]) ((stk, m, i, o)))),
pr2 (***ME*** ([**loadn**,**0**]) ((stk, m, i, o))),
pr3 (***ME*** ([**loadn**,**0**]) ((stk, m, i ,o))),
pr4 (***ME*** ([**loadn**,**0**]) ((stk, m, i, o)))>>)
= <0, <stk, m, i, o>>

Proof in T_{target}

(***ME*** ([**loadn**, **0**]) ((stk, m, i, o)) = error → error,
<***hd*** (pr1 (***ME*** ([**loadn**,**0**]) ((stk, m, i, o)))),
<***tl***(pr1(***ME*** ([**loadn**,**0**]) ((stk, m, i, o)))),
pr2 (***ME*** ([**loadn**,**0**]) ((stk, m, i, o))),
pr3 (***ME*** ([**loadn**,**0**]) ((stk, m, i ,o))),
pr4 (***ME*** ([**loadn**,**0**]) ((stk, m, i, o)))>>)

(axioms TE2, TE1, and I1)

=(<0 • stk, m, i, o,> = error → error, <0, <stk, m, i, o >>)

(conditional axiom)

= <0, <stk, m, i, o, >>

Axiom (E1b)

Translation and proof are similar to those for Axiom (E1a).

Axiom (E2a)

Translation and proof are similar to those for Axiom (E1a).

Axiom (E2b)

Translation and proof are similar to those for Axiom (E1a).

Axiom (E3)

Translate Axiom into L_{target}

E ⟦ **read** ⟧ (<m, i, o>) = null(i) → error,
<(***hd***(i)), (m, ***tl***(i),o)>

(translate axiom using interpretation, ***I***)

λEk. [***H*** (E) (k)] ⟦ ***I***(**read**) ⟧ ***I***(<m, i, o>) = ***I*** [null(i) → error,
<(***hd***(i)), (m, ***tl***(i),o)>)]

(simplify)

H (**[read]**) (<stk, m, i, o>) = (null(i) → error, <***hd*** (i) • stk, m, ***tl***(i), o>)

(simplify)

(***ME*** ([**read**]) ((stk, m, i, o,)) = error → error,
<***hd*** (pr1(***ME*** ([**read**]) ((stk, m, i, o)))),
<***tl*** (pr1(***ME*** ([**read**]) ((stk, m, i, o)))),
pr2 (***ME*** ([**read**]) ((stk, m, i, o))),
pr3 (***ME*** ([**read**]) ((stk, m, i, o))),
pr4 (***ME*** ([**read**]) ((stk, m, i, o)))>>)
= (null(i) → error, <***hd*** (i) • stk, m, ***tl***(i), o))

Proof in T_{target}

(***ME*** (**[read]**) ((stk, m, i, o,)) = error → error,
<***hd*** (pr1(***ME*** (**[read]**) ((stk, m, i, o)))),
<***tl*** (pr1(***ME*** (**[read]**) ((stk, m, i, o)))),
pr2 (***ME*** (**[read]**) ((stk, m, i, o))),
pr3 (***ME*** (**[read]**) ((stk, m, i, o))),
pr4 (***ME*** (**[read]**) ((stk, m, i, o)))>>)

(axioms TE2, TE1, and I5)

= (null(i) → error, <***hd*** (i) • stk, m, ***tl***(i), o>)

Axiom (E4)

Translation and proof are similar to those for Axiom (E3).

Axiom (E5)

Translate Axiom into L_{target}

E ⟦ **not** E ⟧ (s) =
(***E*** ⟦ E ⟧ s = <v, s'>) → [is-bool(v) → <~v, s'>, error], error

(translate axiom using interpretation, ***I***)

λEk. [***H*** (E) (k)] ⟦ ***I***(**not** E) ⟧ ***I***(s) =
(λEk. [***H*** (E) (k)] ⟦ ***I*** (E) ⟧ ***I*** (s) = <v, ***I*** (s')>) →
[is-bool(v) → <~v, I(s')>, error], error

(simplify)

H (***I***(E) • **[not]**) (<stk, m, i, o>) =
(***H*** ⟦ ***I*** (E) ⟧ (<stk, m, i, o >) = <v, <stk', m', i', o'>>) → [is-bool (v) →
<~v, <stk', m', i', o'>>, error], error

(simplify)

F (***I***(E) • **[not]**)
= (F(***I*** (E)) = <v, <stk', m', i', o'>>) → [is-bool(v) →
<~v, <stk', m', i', o'>>, error], error
where F(x) is
(***ME*** (x) ((stk, m, i, o)) = error → error,
<***hd*** (pr1 (***ME*** (x) ((stk, m, i, o)))),
<***tl*** (pr1 (***ME*** (x) ((stk, m, i, o)))),
pr2 (***ME*** (x) ((stk, m, i, o))),
pr3 (***ME*** (x) ((stk, m, i, o))),
pr4 (***ME*** (x) ((stk, m, i, o>>)))>>)

(simplify)

F (***I***(E) • **[not]**)
=(***ME*** (***I*** (E)) ((stk m, i, o)) = error) → error,
let *ME* (***I*** (E)) ((stk, m, i, o)) = <v•stk, m, i, o> **in**
[is-bool (v) → <~v, <stk, m, i, o>>, error]

Proof in T_{target}

F (***I***(E) • **[not]**)

(definition of F)

= (G = error → error, <***hd*** (pr1(G)), <***tl*** (pr1(G)), pr2(G), pr3(G) pr4(G)>>)
where G is ***ME*** (***I***(E) • **[not]**) ((stk, m, i, o))

(axioms TE3, TE2, TE1, and I7)

= (G = error → error, <***hd*** (pr1(G)), <***tl*** (pr1(G)), pr2(G), pr3(G) pr4(G)>>)
where G is
ME (I(E)) ((stk, m, i, o)) = error → error,
let *ME* (***I*** (E)) ((stk, m, i, o)) = (stk', m', i', o') **in**
[***lg*** (stk') <1 → error,
(is-bool(***hd*** (stk')) → (~***hd***(stk') • ***tl*** (stk') m' i', o',), error)]

(STACK-HAS-ONE lemma)

= (G = error → error, <***hd*** (pr1(G)), <***tl*** (pr1(G), pr2(G), pr3(G) pr4(G)>>)
where G is
ME (I(E) ((stk, m, i, o)) = error → error,
let *ME* (***I*** (E)) ((stk, m, i, o)) = (stk', m', i', o')) **in**
(is-bool(***hd*** (stk')) → (~***hd***(stk') • ***tl*** (stk') m' i', o',), error)

(simplify)

=(**ME** (**I** (E)) ((stk m, i, o)) = error) → error,
let ME (**I** (E)) ((stk, m, i, o)) = (stk′, m′, i′, o′)) **in**
[is-bool(**hd**(stk′)) → <~**hd**(stk′), <**tl**(stk′), m′, i′, o′>>, error]

Lemmas

STACK-NOT-EMPTY Lemma

ME(**I**(E))((stk, m, i, o)) = error, or <v • stk, m, i′, o>

Proof

E ∈ {**0**, **1**, **true**, **false**, **read**, I_1, **not** E_1, E_1 = E_2, E_1 + E_2}
where E_1: exp and E_2: exp

Basis

E ∈ {**0**, **1**, **true**, **false**, **read**, I_1}

∴ **I**(E) ∈ {[**loadn**, 0], [**loadn**, 1], [**loadb**, TRUE], [**loadb**, FALSE], [**read**], [**load**, I_1]}

∴ **ME** (**I**(E)) (<stk, m, i, o>)

(TE2 and TE1 axioms)

=**MI** (**I**(E)) <stk, m, i, o>

(definition of **MI**)

=error, or <v • stk, m, i′, o>

where v ∈ {0, 1, tt, ff, **hd**(i), m(I_1)}
and i′ = **tl**(i), if **I**(E)=[**read**]
i, otherwise

Induction step, assume property true for all expression constituents of the expression

E ∈ {**not** E_1, E_1 = E_2, E_1 + E_2}

ME(**I**(E_1) • [**not**]) (<stk, m, i, o>)

(TE3 and TE1 axioms)

$= \boldsymbol{MI}([\textbf{not}])(\boldsymbol{ME}(\boldsymbol{I}(E_1))(<stk, m, i, o>))$, or error

(assumption)

$= \boldsymbol{MI}([\textbf{not}])\ (<v> \bullet stk,m,i',o)$, or error

(definition of ***MI***)

$= (<\sim v> \bullet stk,m,i',o)$, or error

$\boldsymbol{ME}(\boldsymbol{I}(E_1) \bullet \boldsymbol{I}(E_2) \bullet [\textbf{eq}])\ (<stk, m, i, o>)$

(TE3 and TE1 axioms)

$= \boldsymbol{MI}([\textbf{eq}])\ (\boldsymbol{ME}(\boldsymbol{I}(E_1) \bullet \boldsymbol{I}(E_2)(<stk, m, i, o>))$, or error

(assumption)

$= \boldsymbol{MI}([\textbf{eq}])\ ((<v_1> \bullet <v_2> \bullet stk,m,i',o))$, or error

(definition of ***MI***)

$= (<v_1=v_2> \bullet stk,m,i,o)$, or error

$\boldsymbol{ME}(\boldsymbol{I}(E_1) \bullet \boldsymbol{I}(E_2) \bullet [\textbf{add}])\ (<stk, m, i, o>)$

(TE3 and TE1 axioms)

$= \boldsymbol{MI}([\textbf{add}])\ (\boldsymbol{ME}(\boldsymbol{I}(E_1) \bullet \boldsymbol{I}(E_2)\ (<stk, m, i, o>))$, or error

(assumption)

$= \boldsymbol{MI}([\textbf{add}])\ ((<v_1> \bullet <v_2> \bullet stk,m,i',o))$, or error

(definition of ***MI***)

$= (<v_1+v_2> \bullet stk,m,i',o)$, or error

STACK-HAS-ONE Lemma

$\boldsymbol{ME}\ (\boldsymbol{I}\ (E))\ ((stk,\ m,\ i,\ o)) = error \rightarrow error,$
$[\boldsymbol{lg}(pr1(\boldsymbol{ME}(\boldsymbol{I}(E))((stk,\ m,\ i,\ o))\ ))\ <\ 1 \rightarrow A,\ B]$

where E: exp and A, B: (mstate ⊕ {error})

can be rewritten as

$\boldsymbol{ME}\ (\boldsymbol{I}\ (E))\ ((stk,\ m,\ i,\ o)) = error \rightarrow error,\ B$

Proof

$\boldsymbol{ME}\ (\boldsymbol{I}\ (E))\ ((stk,\ m,\ i,\ o)) = error \rightarrow error,$
$[\boldsymbol{lg}(pr1(\boldsymbol{ME}(\boldsymbol{I}(E))((stk,\ m,\ i,\ o))\ ))\ <\ 1 \rightarrow A,\ B]$

(STACK-NOT-EMPTY lemma)

$\boldsymbol{ME}\ (\boldsymbol{I}\ (E))\ ((stk,\ m,\ i,\ o)) = error \rightarrow error,\ [\boldsymbol{lg}\ (v \bullet stk) < 1 \rightarrow A,\ B]$

(conditional simplification)

$\boldsymbol{ME}\ (\boldsymbol{I}\ (E))\ ((stk,\ m,\ i,\ o)) = error \rightarrow error,\ B$

Appendix D

Implementation of CS-Tiny

Theory for Source Language, T_{source}

Language for Source Language, L_{source}

Language Elements	Defined Language	Defining Language
domains	id = {**I**, $\mathbf{I_1}$, $\mathbf{I_2}$...}	num
	exp	bool
	com	value = num ⊕ bool
	prog	input = value*
		output = value*
		mem = id → [value ⊕ {**unbound**}]
		state = mem ⊗ input ⊗ output
		error = {**empty-input**, **unbound-var**, **non-bool-value**, **non-num-value** **empty-stk-error**}
		ans = state ⊕ error
		cont = state → ans
		econt = value → cont

function symbols	**0**: → exp **1**: → exp **true**: → exp **false**: → exp **read**: → exp {**I**, $\mathbf{I_1}$, $\mathbf{I_2}$...}: → exp **not**: exp → exp =: exp ⊗ exp → exp +: exp ⊗ exp → exp :=: id ⊗ exp → com **output**: exp → com **if**: exp ⊗ com ⊗ com → com **while**: exp ⊗ com → com ;: com ⊗ com → com **begin**: com → prog	***E***: exp → econt → cont ***C***: com → cont → cont ***P***: prog → input → [output ⊕ error] ***hd***: value* → value ⊕ error ***tl***: value* → value* _ • _: value ⊗ value* → value* _+_: num ⊗ num → num
predicate symbols		null: value* → bool ∈ error: ans → bool
individual variable symbols		k: econt v, v′, v_i: value, $1 \leq i \leq n$ m: mem i: input o: output C, C_1, C_2: com E, E_1, E_2: exp c_o, c: cont a_1, a_2: ans I, I′: id v*: value* s: state P: prog (m,i,o): state b: bool

Axioms for Source Language, A_{source}

(E1a) $\boldsymbol{E}$ [[**0**]] (k) $=_{econt}$ k(0)

(E1b) $\boldsymbol{E}$ [[**1**]] (k) $=_{econt}$ k(1)

(E2a) $\boldsymbol{E}$ [[**true**]] $(k) =_{econt} k(\text{TRUE})$

(E2b) $\boldsymbol{E}$ [[**false**]] $(k) =_{econt} k(\text{FALSE})$

(E3) $\boldsymbol{E}$ [[**read**]] $(k) =_{econt} \lambda(m,i,o).\ [\text{null}(i) \rightarrow$ **empty-input**, $k(\boldsymbol{hd}(i))((m, \boldsymbol{tl}(i), o))]$

(E4) $\boldsymbol{E}$ [[I]] $(k) =_{econt} \lambda(m,i,o).\ [m(I) =$ **unbound** $\rightarrow$ **unbound-var**, $k(m(I))(v_1)\ldots(v_n)((m,i,o))]$

(E5) $\boldsymbol{E}$ [[**not** E]] $(k) =_{econt}$ $\boldsymbol{E}$ [[E]] $(\lambda vs.\ [\text{is-bool}(v) \rightarrow k(\sim v)(s),$ **non-bool-value**$])$

(E6) $\boldsymbol{E}$ [[$E_1 = E_2$]] $(k) =_{econt}$ $\boldsymbol{E}$ [[E_1]] $(\lambda v'.$ $\boldsymbol{E}$ [[E_2]] $(\lambda v.\ [k(v =_{value} v')]))$

(E7) $\boldsymbol{E}$ [[$E_1 + E_2$]] $(k) =_{econt}$ $\boldsymbol{E}$ [[E_1]] $(\lambda v'.$ $\boldsymbol{E}$ [[E_2]] $(\lambda v.\ [\text{is-num}(v)\ \&\ \text{is-num}(v') \rightarrow$
$k(v+v')$,
non-num-value$]))$

(C1) $\boldsymbol{C}$ [[I := E]] $(c) =_{cont}$ $\boldsymbol{E}$ [[E]] $(\lambda v(m,i,o).\ [c((m[v/I],i,o))])$

(C2) $\boldsymbol{C}$ [[**output** E]] $(c) =_{cont}$ $\boldsymbol{E}$ [[E]] $(\lambda v(m,i,o).\ [c((m,i,o \bullet \langle v \rangle))])$

(C3) $\boldsymbol{C}$ [[**if** E C_1 C_2]] $(c) =_{cont}$
$\boldsymbol{E}$ [[E]] $(\lambda v.\ [\text{is-bool}(v) \rightarrow (v \rightarrow$ $\boldsymbol{C}$ [[C_1]] (c), $\boldsymbol{C}$ [[C_2]] $(c))$, **non-bool-value**$])$

(C4) $\boldsymbol{C}$ [[**while** E C]] $(c) =_{cont}$
$\boldsymbol{E}$ [[E]] $(\lambda v.\ [\text{is-bool}(v) \rightarrow (v \rightarrow$ $\boldsymbol{C}$ [[C]] $($$\boldsymbol{C}$ [[**while** E C]] $(c)), c)$,
non-bool-value$])$

(C5) $\boldsymbol{C}$ [[$C_1 ; C_2$]] $(c) =_{cont}$ $\boldsymbol{C}$ [[C_1]] $($ $\boldsymbol{C}$ [[C_2]] $(c)\ )$

(P1) $\boldsymbol{P}$ [[**begin** P]] $(i) =_{output \oplus error} \lambda a.\ [a \in \text{error} \rightarrow a,\ \boldsymbol{hd}(\boldsymbol{tl}(\boldsymbol{tl}(a)))]$
$($ $\boldsymbol{C}$ [[P]] $(c_o)\ (m_o, i, \langle\rangle))$
where $c_o = \lambda s.s$
$\forall I \in \text{id},\ m_o(I) =$ **unbound**
$\langle\rangle$ = initially empty output

(A1) $m[v/I](I') =_{value \oplus \{\textbf{unbound}\}} (I =_{id} I' \rightarrow v,\ m(I'))$

(A2a) $\boldsymbol{hd}(\langle\rangle) =_{error \oplus value}$ **empty-stk-error**

(A2b) $\boldsymbol{hd}(\langle v \rangle) =_{error \oplus value} v$

(A2c) $\boldsymbol{hd}(\langle v \rangle \bullet v^*) =_{error \oplus value} v$

(A2d) $\boldsymbol{tl}(\langle v \rangle \bullet v^*) = v^*$

(A2e) $\boldsymbol{hd}(v^*) \bullet \boldsymbol{tl}(v^*) = v^*$

Theory for Target Language, T_{target}

Language for Target Language, L_{target}

Language Elements	Defined Language	Defining Language
domains	id instr code = instr* ecode = instr* num bool pcode = instr*	value = num ⊕ bool stack = value* mem = id → [value ⊕ {**unbound**}] input = value* output = value* state = mem ⊗ input ⊗ output mstate = stack ⊗ state mans = mstate ⊕ error mcont = mstate → mans error = {**empty-input**, **unbound-var**, **non-bool-value**, **non-num-value** **empty-stk-error**, **stack-underflow**}
function symbols	**start**: → instr **halt**: → instr **loadn**: num → instr **loadb**: bool → instr **read**: → instr **load**: id → instr **not**: → instr **eq**: → instr **add**: → instr **store**: id → instr **output**: → instr **cond**: code ⊗ code → instr **loop**: ecode ⊗ code → instr	***MI***: instr → mcont → mcont ***MC***: code → mcont → mcont ***ME***: ecode → mcont → mcont ***MP***: pcode → mcont → mcont ***hd***: value* → value ⊕ error ***tl***: value* → value* ***lg***: value* → num _ • _: value ⊗ value* → value* _+_: num ⊗ num → num
predicate symbols		null: value* → bool <: num ⊗ num → bool

individual variable symbols	z: mcont stk: stack m: mem i: input o: output (stk,m,i,o): mstate ID: id P, Q: code T: ecode I: instr v: value

Axioms for Target Language, A_{target}

(I1) ***MI*** ⟦ [**loadn**, 0] ⟧ (z) ((stk,m,i,o)) $=_{mans}$ z((<0>•stk,m,i,o))

(I2) ***MI*** ⟦ [**loadn**, 1] ⟧ (z) ((stk,m,i,o)) $=_{mans}$ z((<1>•stk,m,i,o))

(I3) ***MI*** ⟦ [**loadb**, TRUE] ⟧ (z) ((stk,m,i,o)) $=_{mans}$ z((<TRUE>•stk,m,i,o))

(I4) ***MI*** ⟦ [**loadb**, FALSE] ⟧ (z) ((stk,m,i,o)) $=_{mans}$ z((<FALSE>•stk,m,i,o))

(I5) ***MI*** ⟦ [**read**] ⟧ (z) ((stk,m,i,o)) $=_{mans}$
[null(i) → **empty-input**,
z((<***hd***(i)>•stk,m,***tl***(i),o))]

(I6) ***MI*** ⟦ [**load**,ID] ⟧ (z) ((stk,m,i,o)) $=_{mans}$
[m(ID) $=_{value}$ **unbound** → **unbound-var**,
z((<m(ID)>•stk,m,i,o))]

(I7) ***MI*** ⟦ [**not**] ⟧ (z) ((stk,m,i,o)) $=_{mans}$
[***lg***(stk) < 1 → **stack-underflow**,
(is-bool(***hd***(stk)) → z((<~***hd***(stk)>•***tl***(stk),m,i,o)), **non-bool-value**)]

(I8) ***MI*** ⟦ [**eq**] ⟧ (z) ((stk,m,i,o)) $=_{mans}$
[***lg***(stk) < 2 → **stack-underflow**,
z((<***hd***(***tl***(stk)) = ***hd***(stk)>•***tl***(***tl***(stk)),m,i,o))]

(I9) ***MI*** ⟦ [**add**] ⟧ (z) ((stk,m,i,o)) $=_{mans}$
[***lg***(stk) < 2 → **stack-underflow**,
(is-num(***hd***(stk)) & is-num(***hd***(***tl***(stk))) →
z((<***hd***(***tl***(stk)) + ***hd***(stk)>•***tl***(***tl***(stk)),m,i,o)),
non-num-value)]

(I10) $\boldsymbol{MI} \llbracket [\mathbf{store}, ID] \rrbracket (z)\ ((stk,m,i,o)) =_{mans}$

$$[\boldsymbol{lg}(stk) < 1 \rightarrow \mathbf{stack\text{-}underflow}, \\ \quad z((\boldsymbol{tl}(stk), m[\boldsymbol{hd}(stk)/ID], i, o))]$$

(I11) $\boldsymbol{MI} \llbracket [\mathbf{output}] \rrbracket (z)\ ((stk,m,i,o)) =_{mans}$

$$[\boldsymbol{lg}(stk) < 1 \rightarrow \mathbf{stack\text{-}underflow}, \\ \quad z((\boldsymbol{tl}(stk), m, i, o\bullet\langle \boldsymbol{hd}(stk)\rangle))]$$

(I12) $\boldsymbol{MI} \llbracket [\mathbf{cond}, P, Q] \rrbracket (z)\ ((stk,m,i,o)) =_{mans}$

$$[\boldsymbol{lg}(stk) < 1 \rightarrow \mathbf{stack\text{-}underflow}, \\ \quad [is\text{-}bool(\boldsymbol{hd}(stk)) \rightarrow \\ \quad\quad [\boldsymbol{hd}(stk) \rightarrow \boldsymbol{MC} \llbracket P \rrbracket (z)\ ((\boldsymbol{tl}(stk),m,i,o)), \\ \quad\quad\quad \boldsymbol{MC} \llbracket Q \rrbracket (z)\ ((\boldsymbol{tl}(stk),m,i,o))], \\ \quad\quad \mathbf{non\text{-}bool\text{-}value}]]$$

(I13) $\boldsymbol{MI} \llbracket [\mathbf{loop}, T, P] \rrbracket (z)\ (stk,m,i,o) =_{mans}$

$$\boldsymbol{ME} \llbracket T \rrbracket (\lambda(stk,m,i,o).\ [is\text{-}bool(\boldsymbol{hd}(stk)) \rightarrow \\ \quad [\boldsymbol{hd}(stk) \rightarrow \\ \quad\quad \boldsymbol{MC} \llbracket P \rrbracket (\boldsymbol{MI} \llbracket [\mathbf{loop}, T, P] \rrbracket (z))\ ((\boldsymbol{tl}(stk),m,i,o)), \\ \quad\quad z((\boldsymbol{tl}(stk),m,i,o))], \\ \quad \mathbf{non\text{-}bool\text{-}value}])$$

(I14) $\boldsymbol{MI} \llbracket [\mathbf{start}] \rrbracket (z)\ ((stk,m,i,o)) =_{mans} z((\langle\rangle, m_0, i, \langle\rangle))$

(I15) $\boldsymbol{MI} \llbracket [\mathbf{halt}] \rrbracket (z)\ ((stk,m,i,o)) =_{mans} (stk,m,i,o)$

(TC1) $\boldsymbol{MC} \llbracket \langle\rangle \rrbracket (z) =_{mcont} z$

(TC2) $\boldsymbol{MC} \llbracket \langle I\rangle\bullet P \rrbracket (z) =_{mcont} \boldsymbol{MI} \llbracket I \rrbracket (\boldsymbol{MC} \llbracket P \rrbracket (z))$

(TC3) $\boldsymbol{MC} \llbracket P\bullet Q \rrbracket (z) =_{mcont} \boldsymbol{MC} \llbracket P \rrbracket (\boldsymbol{MC} \llbracket Q \rrbracket (z))$

(TE1) $\boldsymbol{ME} \llbracket \langle\rangle \rrbracket (z) =_{mcont} z$

(TE2) $\boldsymbol{ME} \llbracket \langle I\rangle\bullet P \rrbracket (z) =_{mcont} \boldsymbol{MI} \llbracket I \rrbracket (\boldsymbol{ME} \llbracket P \rrbracket (z))$

(TE3) $\boldsymbol{ME} \llbracket P\bullet Q \rrbracket (z) =_{mcont} \boldsymbol{ME} \llbracket P \rrbracket (\boldsymbol{ME} \llbracket Q \rrbracket (z))$

(TP1) $\boldsymbol{MP} \llbracket \langle\rangle \rrbracket (z) =_{mcont} z$

(TP2) $\boldsymbol{MP} \llbracket \langle I\rangle\bullet P \rrbracket (z) =_{mcont} \boldsymbol{MI} \llbracket I \rrbracket (\boldsymbol{MP} \llbracket P \rrbracket (z))$

(TP3) $\boldsymbol{MP} \llbracket P\bullet Q \rrbracket (z) =_{mcont} \boldsymbol{MP} \llbracket P \rrbracket (\boldsymbol{MP} \llbracket Q \rrbracket (z))$

(A1) $m[v/I](I') =_{value \oplus \{\mathbf{unbound}\}} [I =_{id} I' \rightarrow v, m(I')]$

(A2a) $\boldsymbol{hd}(\langle\rangle) =_{error \oplus value} \mathbf{empty\text{-}stk\text{-}error}$

(A2b) $\boldsymbol{hd}(\langle v\rangle) =_{error \oplus value} v$

(A2c) $\boldsymbol{hd}(\langle v\rangle\bullet v^*) =_{error \oplus value} v$

(A2d) $\boldsymbol{tl}(<v>\bullet v^*) = v^*$

(A2e) $\boldsymbol{hd}(v^*)\bullet \boldsymbol{tl}(v^*) = v^*$

Interpretation

Language Elements	Defined Language	Defining Language
$\mathcal{I}$: domain → domain	$\mathcal{I}$(id) = id $\mathcal{I}$(exp) = ecode $\mathcal{I}$(com) = code $\mathcal{I}$(prog) = pcode	$\mathcal{I}$(num) = num $\mathcal{I}$(bool) = bool $\mathcal{I}$(value) = value $\mathcal{I}$(input) = mstate $\mathcal{I}$(output) = mstate $\mathcal{I}$(mem) = mem $\mathcal{I}$(state) = mstate $\mathcal{I}$(state$_o$) = mstate$_o$ $\mathcal{I}$(ans) = mans $\mathcal{I}$(cont) = mcont $\mathcal{I}$(econt) = value → mstate → mans $\mathcal{I}$(error) = error
$\mathcal{I}$: function symbol → term	$\mathcal{I}$(**0**) = [**loadn**, 0]	$\mathcal{I}$(***E***) = λEk. ***H*** (E) (k) where ***H***(C)(λv(stk, m, i, o). F) equals ***ME***(C)(λ(stk, m, i, o). F(***hd***(stk))((***tl***(stk), m, i, o))
	$\mathcal{I}$(**1**) = [**loadn**, 1] $\mathcal{I}$(**true**) = [**loadb**, TRUE]	$\mathcal{I}$(***C***) = λCc. ***MC*** (C) (c) $\mathcal{I}$(***P***) = λP. ***MP*** (P) z_o, where z_o = λ(stk,m,i,o). (stk,m,i,o)
	$\mathcal{I}$(**false**) = [**loadb**, FALSE] $\mathcal{I}$(**read**) = [**read**] $\mathcal{I}$(**I_i**) = [**load**, **I_i**], $i \geq 1$ $\mathcal{I}$(**not**) = λE. (E) • [**not**] $\mathcal{I}$(**=**) = λE_1E_2. (E_1) • (E_2) • [**eq**] $\mathcal{I}$(**+**) = λE_1E_2. (E_1) • (E_2) • [**add**] $\mathcal{I}$(**:=**) = λIE. (E) • [**store**, I] $\mathcal{I}$(**output**) = λE. (E) • [**output**] $\mathcal{I}$(**if**) = λEC_1C_2. (E) • [**cond**, C_1, C_2] $\mathcal{I}$(**while**) = λEC. [**loop**, E, C] $\mathcal{I}$(**;**) = λC_1C_2. (C_1) • (C_2) $\mathcal{I}$(**begin**) = λC. [**start**] • (C) • [**halt**]	$\mathcal{I}$(***hd***) = λs. ***hd***(s) $\mathcal{I}$(***tl***) = λs. ***tl***(s)

predicate symbol → predicate symbol	$I(=_{econt}) = =_{mcont}$ $I(=_{ans}) = =_{mans}$ $I(=_{value}) = =_{value}$ $I(=_{id}) = =_{id}$ I(null) = null
individual variable symbol → term	I(k: econt) = λv(stk, m, i, o). (z)((v • stk, m, i, o)): (value → mstate → mans) I((m,i,o): state) = (stk,m,i,o): mstate I(s: state) = (stk,m,i,o): mstate I(c: cont) = z: mcont I((m_o,i,<>): initial state) = (<>,m_o,i,<>): initial mstate I(c_o: cont) = z_o: mcont I(i: input) = (stk,m,i,o): mstate I(***hd***(***tl***(***tl***(m, i, o)))): output) = (stk, m, i, o): mstate
new predicates	is-econt: (value → mstate → mans) → bool is-cont: mcont → bool is-ans: mans → bool is-state: mstate → bool is-input: mcont → bool N.B., there are other predicates. They are all trivially true.

Example Correctness Proof

Axiom (E1a)

Translate Axiom into L_{target}

$\mathcal{E}$ ⟦ **0** ⟧ (k) $=_{econt}$ k(0)

(translate axiom using interpretation, *I*)

λEk. [*H* (E) (k)] (*I*(**0**)) (*I*(k)) $=_{mcont}$ *I*(k)(*I*(0))

(simplify)

H ([**loadn**, **0**]) (λv(stk, m, i, o). z((v•stk, m, i, o))) $=_{mcont}$
(λv(stk, m, i, o). z((v•stk, m, i, o)))(0)

(simplify)

ME ([**loadn**, 0]) (z) $=_{mcont}$ λ(stk,m,i,o). [z((0 • stk,m,i,o))]

Proof in T_{target}

ME ([**loadn**, 0]) (z)

(axiom TE2)

= ***MI*** ⟦ [**loadn**, 0] ⟧ (***ME*** ⟦ <> ⟧ (z))

(axiom TE1)

=***MI***([**loadn**, 0])z

(axiom I1)

=λ(stk,m,i,o). [z((<0>•stk,m,i,o))]

Axiom (E1b)

Translation and proof are similar to those for Axiom (E1a).

Axiom (E2a)

Translation and proof are similar to those for Axiom (E1a).

Axiom (E2b)

Translation and proof are similar to those for Axiom (E1a).

Axiom (E3)

Translate Axiom into L_{target}

E ⟦ **read** ⟧ (k) $=_{econt}$ λ(m,i,o). [null(i) → **empty-input**,
k(***hd***(i))((m,***tl***(i),o))]

(translate axiom using interpretation, ***I***)

λEk. [***H*** (E) (k)] ⟦ ***I***(**read**) ⟧ ***I***(k) $=_{mcont}$ ***I***(λ(m,i,o). [null(i) → **empty-input**,
k(***hd***(i))((m,***tl***(i),o))])

(simplify)

ME ([**read**]) (z) $=_{mcont}$ λ(stk, m, i, o). [null(i) → **empty-input**,
z((***hd***(i)•stk, m, ***tl***(i), o))]

Proof in T_{target}

ME ([**read**]) (z)

(axiom TE2)

= ***MI*** ⟦ [**read**] ⟧ (***ME*** ⟦ <> ⟧ (z))

(axiom TE1)

= ***MI*** ⟦ [**read**] ⟧ z

(axiom I5)

= λ(stk,m,i,o). [null(i) → **empty-input**,
z((<**hd**(i)>•stk,m,**tl**(i),o))]

Axiom (E4)

Translation and proof are similar to those for Axiom (E3).

Axiom (E5)

Translate Axiom into L_{target}

E ⟦ **not** E ⟧ (k) $=_{econt}$ ***E*** ⟦ E ⟧ (λvs. [is-bool(v) →
k(~v)(s),
non-bool-value])

(translate axiom using interpretation, ***I***)

λEk. [***H*** (E) (k)] ⟦ ***I***(**not** E) ⟧ ***I***(k) $=_{mcont}$ λEk. [***H*** (E) (k)] ⟦ ***I***(E) ⟧ (***I*** (λvs. [is-bool(v) →
k(~v)(s),
non-bool-value]))

(simplify)

ME (***I***(E) • [**not**]) (z) $=_{mcont}$ ***ME*** (***I***(E)) (λ(<v>•stk,m,i,o). [is-bool(v) →
z((<~v>•stk,m,i,o)),
non-bool-value])

Proof in T_{target}

ME (***I***(E) • [**not**]) (z)

(axiom TE3)

= ***ME*** ⟦ ***I***(E) ⟧ (***ME*** ⟦ [**not**] ⟧ (z))

(axiom TE2)

= ***ME*** ⟦ ***I***(E) ⟧ (***MI*** ⟦ [**not**] ⟧ (***ME*** ⟦ <> ⟧ (z)))

(axiom TE1)

= ***ME*** ⟦ ***I***(E) ⟧ (***MI*** ⟦ [**not**] ⟧ (z))

(axiom I7)

$= \boldsymbol{ME} ⟦ \boldsymbol{I}(E) ⟧ (\lambda(stk,m,i,o).\ [\boldsymbol{lg}(stk) < 1 \rightarrow$ **stack-underflow**,
$(is\text{-}bool(\boldsymbol{hd}(stk)) \rightarrow$
$z((<\sim\boldsymbol{hd}(stk)> \bullet \boldsymbol{tl}(stk),m,i,o)),$
non-bool-value]])

(STACK-HAS-ONE lemma)

$= \boldsymbol{ME}\ (\boldsymbol{I}(E))\ (\lambda(stk,m,i,o).\ [is\text{-}bool(\boldsymbol{hd}(stk)) \rightarrow$
$z((<\sim\boldsymbol{hd}(stk)> \bullet \boldsymbol{tl}(stk),m,i,o)),$
non-bool-value])

Axiom (E6)

Translate Axiom into L_{target}

$\boldsymbol{E} [\![E_1 = E_2]\!] (k) =_{econt} \boldsymbol{E} [\![E_1]\!] (\lambda v_1 . \boldsymbol{E} [\![E_2]\!] (\lambda v_2 s. [k(v_1 =_{value} v_2)(s)]))$

(translate axiom using interpretation, $\boldsymbol{I}$)

$\lambda Ek. [\boldsymbol{H}(E)(k)] [\![\boldsymbol{I}(E_1 = E_2)]\!] \boldsymbol{I}(k) =_{mcont} \lambda Ek. [\boldsymbol{H}(E)(k)] [\![\boldsymbol{I}(E_1)]\!]$
$(\boldsymbol{I}(\lambda v_1 . \boldsymbol{E} [\![E_2]\!] (\lambda v_2 s. k(v_1 = v_2)(s))))$

(simplify)

$\boldsymbol{ME}(\boldsymbol{I}(E_1) \bullet \boldsymbol{I}(E_2) \bullet [\mathbf{eq}])(z) =_{mcont} \boldsymbol{H}(\boldsymbol{I}(E_1))(\lambda v_1 . \boldsymbol{H}(\boldsymbol{I}(E_2))$
$(\lambda v_2 (stk, m, i, o). z((<v_1 = v_2> \bullet stk, m, i, o))))$

(simplify)

$\boldsymbol{ME}(\boldsymbol{I}(E_1) \bullet \boldsymbol{I}(E_2) \bullet [\mathbf{eq}])(z) =_{mcont} \boldsymbol{ME}(\boldsymbol{I}(E_1))(\lambda(stk', m', i', o'). \boldsymbol{ME}(\boldsymbol{I}(E_2))($
$\lambda(stk,m,i,o). [z((<\boldsymbol{hd}(stk') =_{value} \boldsymbol{hd}(stk)> \bullet \boldsymbol{tl}(stk),m,i,o))])((\boldsymbol{tl}(stk'),m',i',o')))$

(STACK-NOT-EMPTY lemma)

$\boldsymbol{ME}(\boldsymbol{I}(E_1) \bullet \boldsymbol{I}(E_2) \bullet [\mathbf{eq}])(z) =_{mcont} \boldsymbol{ME}(\boldsymbol{I}(E_1))(\lambda(stk', m', i', o').$
$(\lambda(stk,m,i,o). [z((<\boldsymbol{hd}(stk') =_{value} \boldsymbol{hd}(stk)> \bullet \boldsymbol{tl}(stk),m,i,o))])((v_2 \bullet \boldsymbol{tl}(stk'),m',i',o')))$
where $\boldsymbol{ME}(\boldsymbol{I}(E_2))(z)(\boldsymbol{tl}(stk'), m', i', o') = z(v_2 \bullet \boldsymbol{tl}(stk'), m', i', o')$

(simplify)

$\boldsymbol{ME}(\boldsymbol{I}(E_1) \bullet \boldsymbol{I}(E_2) \bullet [\mathbf{eq}])(z) =_{mcont} \boldsymbol{ME}(\boldsymbol{I}(E_1))(\lambda(stk', m', i', o').$
$z((<\boldsymbol{hd}(stk') =_{value} v_2> \bullet \boldsymbol{tl}(stk'),m',i',o')))$
where $\boldsymbol{ME}(\boldsymbol{I}(E_2))(z)(\boldsymbol{tl}(stk'), m', i', o') = z(v_2 \bullet \boldsymbol{tl}(stk'), m', i', o')$

(STACK-NOT-EMPTY lemma)

$\boldsymbol{ME}(\boldsymbol{I}(E_1) \bullet \boldsymbol{I}(E_2) \bullet [\mathbf{eq}])(z) =_{mcont} \lambda(stk', m', i', o').$
$z((<v_1 =_{value} v_2> \bullet \boldsymbol{tl}(stk'),m',i',o'))$
where $\boldsymbol{ME}(\boldsymbol{I}(E_2))(z)(\boldsymbol{tl}(stk'), m', i', o') = z(v_2 \bullet \boldsymbol{tl}(stk'), m', i', o')$
and $\boldsymbol{ME}(\boldsymbol{I}(E_1))(z)(stk', m', i', o') = z(v_1 \bullet stk', m', i', o')$

Proof in T_{target}

$\boldsymbol{ME}\ (\boldsymbol{I}(E_1) \bullet \boldsymbol{I}(E_2) \bullet [\mathbf{eq}])\ (z)$

(axiom TE3)

$= \boldsymbol{ME}\ [\![\ \boldsymbol{I}(E_1)\ ]\!]\ (\boldsymbol{ME}\ [\![\ \boldsymbol{I}(E_2) \bullet [\mathbf{eq}]\ ]\!]\ (z))$

(axiom TE3)

$= \boldsymbol{ME}\ [\![\ \boldsymbol{I}(E_1)\ ]\!]\ (\boldsymbol{ME}\ [\![\ \boldsymbol{I}(E_2)\ ]\!]\ (\boldsymbol{ME}\ [\![\ [\mathbf{eq}]\ ]\!]\ (z)))$

(axiom TE2)

$= \boldsymbol{ME}\ [\![\ \boldsymbol{I}(E_1)\ ]\!]\ (\boldsymbol{ME}\ [\![\ \boldsymbol{I}(E_2)\ ]\!]\ (\boldsymbol{MI}\ [\![\ [\mathbf{eq}]\ ]\!]\ (\boldsymbol{ME}\ [\![\ <>\ ]\!]\ (z))))$

(axiom TE1)

$= \boldsymbol{ME}\ [\![\ \boldsymbol{I}(E_1)\ ]\!]\ (\boldsymbol{ME}\ [\![\ \boldsymbol{I}(E_2)\ ]\!]\ (\boldsymbol{MI}\ [\![\ [\mathbf{eq}]\ ]\!]\ (z)))$

(axiom I8)

$= \boldsymbol{ME}\ [\![\ \boldsymbol{I}(E_1)\ ]\!]\ (\boldsymbol{ME}\ [\![\ \boldsymbol{I}(E_2)\ ]\!]\ (\lambda(stk,m,i,o).\ [\boldsymbol{lg}(stk) < 2 \rightarrow \textbf{stack-underflow},$
$z((<\boldsymbol{hd}(\boldsymbol{tl}(stk)) =_{value} \boldsymbol{hd}(stk)>\bullet \boldsymbol{tl}(\boldsymbol{tl}(stk)),m,i,o))]))$

(STACK-HAS-TWO lemma)

$= \boldsymbol{ME}\ [\![\ \boldsymbol{I}(E_1)\ ]\!]\ (\boldsymbol{ME}\ [\![\ \boldsymbol{I}(E_2)\ ]\!]$
$(\lambda(stk,m,i,o).\ [z((<\boldsymbol{hd}(\boldsymbol{tl}(stk)) =_{value} \boldsymbol{hd}(stk)> \bullet \boldsymbol{tl}(\boldsymbol{tl}(stk)),m,i,o))]))$

(STACK-NOT-EMPTY lemma)

$= \lambda(stk', m', i', o').\ z((<v_1 =_{value} v_2> \bullet \boldsymbol{tl}(stk'),m',i',o'))$
where $\boldsymbol{ME}(\boldsymbol{I}(E_2))(z)(\boldsymbol{tl}(stk'), m', i', o') = z(v_2 \bullet \boldsymbol{tl}(stk'), m', i', o')$
and $\boldsymbol{ME}(\boldsymbol{I}(E_1))(z)(stk', m', i', o') = z(v_1 \bullet stk', m', i', o')$

Axiom (E7)

Translation and proof are similar to those for Axiom (E6).

Axiom (C1)

Translate Axiom into L_{target}

$\mathbf{C} [\![\, I := E \,]\!] \, (c) =_{cont} \boldsymbol{E} [\![\, E \,]\!] \, (\lambda v(m,i,o). \, [c((m[v/I],i,o))])$

(translate axiom using interpretation, $\boldsymbol{I}$)

$\lambda Cc. \, [\boldsymbol{MC} \, (C) \, (c)] \, [\![\, \boldsymbol{I}(I := E) \,]\!] \, \boldsymbol{I}(c) =_{mcont}$
$\lambda Ek. \, [\boldsymbol{H} \, (E) \, (k)] \, [\![\, \boldsymbol{I}(E) \,]\!] \, (\boldsymbol{I} \, (\lambda v(m,i,o). \, [c((m[v/I],i,o))]))$

(translate axiom using interpretation, $\boldsymbol{I}$)

$\boldsymbol{MC} \, (\boldsymbol{I}(E) \bullet [\mathbf{store}, I]) \, (z) =_{mcont}$
$\boldsymbol{H}(\boldsymbol{I}(E))(\lambda v(stk,m,i,o). \, [z((stk,m[v/I],i,o))])$

(simplify)

$\boldsymbol{MC} \, (\boldsymbol{I}(E) \bullet [\mathbf{store}, I]) \, (z) =_{mcont}$
$\boldsymbol{ME} \, (\boldsymbol{I}(E)) \, (\lambda(stk,m,i,o). \, [z((\boldsymbol{tl}(stk),m[\boldsymbol{hd}(stk)/I],i,o))])$

Proof in T_{target}

$\boldsymbol{MC} \, (\boldsymbol{I}(E) \bullet [\mathbf{store}, I]) \, (z)$

(axiom TC3)

$= \boldsymbol{MC} \, [\![\, \boldsymbol{I}(E) \,]\!] \, (\boldsymbol{MC} \, [\![\, [\mathbf{store}, I] \,]\!] \, (z))$

(axiom TC2)

$= \boldsymbol{MC}(\boldsymbol{I}(E)) \, (\boldsymbol{MI} \, [\![\, [\mathbf{store}, I] \,]\!] \, (\boldsymbol{MC} \, [\![\, \langle\rangle \,]\!] \, (z)))$

(axiom TC1)

$= \boldsymbol{MC}(\boldsymbol{I}(E)) \, (\boldsymbol{MI}([\mathbf{store}, I])z)$

(axiom I10)

$= \boldsymbol{MC}(\boldsymbol{I}(E)) \, (\lambda(stk,m,i,o). \, [\boldsymbol{lg}(stk) < 1 \rightarrow \mathbf{stack\text{-}underflow},$
$z((\boldsymbol{tl}(stk),m[\boldsymbol{hd}(stk)/I],i,o))])$

(MC-equals-ME lemma)

$= \boldsymbol{ME}(\mathcal{I}(E))\ (\lambda(stk,m,i,o).\ [\boldsymbol{lg}(stk) < 1 \rightarrow \textbf{stack-underflow},$
$z((\boldsymbol{tl}(stk), m[\boldsymbol{hd}(stk)/I], i, o))])$

(STACK-HAS-ONE lemma)

$= \boldsymbol{ME}\ (\mathcal{I}(E))\ (\lambda(stk,m,i,o).\ [z((\boldsymbol{tl}(stk), m[\boldsymbol{hd}(stk)/I], i, o))])$

Axiom (C2)

Translation and proof are similar to those for Axiom (C1).

Axiom (C3)

Translate Axiom into L_{target}

$$\mathbf{C}\ [\![\ \textbf{if}\ E\ C_1\ C_2\ ]\!]\ (c) =_{cont}\ \mathbf{E}\ [\![\ E\ ]\!]\ (\lambda v.\ [\text{is-bool}(v) \rightarrow (v \rightarrow \mathbf{C}\ [\![\ C_1\ ]\!]\ (c), \mathbf{C}\ [\![\ C_2\ ]\!]\ (c)), \textbf{non-bool-value}])$$

(translate axiom using interpretation, $\mathcal{I}$)

$$\lambda Cc.\ [\boldsymbol{MC}\ (C)\ (c)]\ [\![\ \mathcal{I}(\textbf{if}\ E\ C_1\ C_2)\ ]\!]\ \mathcal{I}(c) =_{mcont}$$
$$\lambda Ek.\ [\boldsymbol{H}\ (E)\ (k)]\ [\![\ \mathcal{I}(E)\ ]\!]$$
$$(\boldsymbol{I}\ (\lambda v(m,i,o).\ [\text{is-bool}(v) \rightarrow (v \rightarrow \mathbf{C}\ [\![\ C_1\ ]\!]\ (c)\ ((m,i,o)), \mathbf{C}\ [\![\ C_2\ ]\!]\ (c)\ ((m,i,o))), \textbf{non-bool-value}]))$$

(simplify)

$$\boldsymbol{MC}\ (\mathcal{I}(E) \bullet [\textbf{cond}, \mathcal{I}(C_1), \mathcal{I}(C_2)])\ (z) =_{mcont}$$
$$\boldsymbol{H}(\mathcal{I}(E))$$
$$(\lambda v(stk,m,i,o).\ [\text{is-bool}(v) \rightarrow (v \rightarrow \boldsymbol{MC}\ [\![\ \mathcal{I}(C_1)\ ]\!]\ (z)\ ((stk,m,i,o)), \boldsymbol{MC}\ [\![\ \mathcal{I}(C_2)\ ]\!]\ (z)\ ((stk,m,i,o))), \textbf{non-bool-value}])$$

(simplify)

$$\boldsymbol{MC}\ (\mathcal{I}(E) \bullet [\textbf{cond}, \mathcal{I}(C_1), \mathcal{I}(C_2)])\ (z) =_{mcont}$$
$$\boldsymbol{ME}\ (\mathcal{I}(E))\ (\lambda(stk,m,i,o).$$
$$\text{is-bool}(\boldsymbol{hd}(stk)) \rightarrow (\boldsymbol{hd}(stk) \rightarrow \boldsymbol{MC}\ [\![\ \mathcal{I}(C_1)\ ]\!]\ (z)\ ((\boldsymbol{tl}(stk),m,i,o)), \boldsymbol{MC}\ [\![\ \mathcal{I}(C_2)\ ]\!]\ (z)\ ((\boldsymbol{tl}(stk),m,i,o))), \textbf{non-bool-value})$$

Proof in T_{target}

$$\boldsymbol{MC}\ (\mathcal{I}(E) \bullet [\textbf{cond},\ \mathcal{I}(C_1),\ \mathcal{I}(C_2)])\ (z)$$

(axiom TC3)

$$= \boldsymbol{MC}\ [\![\ \mathcal{I}(E)\ ]\!]\ (\boldsymbol{MC}\ [\![\ [\textbf{cond},\ \mathcal{I}(C_1),\ \mathcal{I}(C_2)]\ ]\!]\ (z))$$

(axiom TC2)

$$= \boldsymbol{MC}\ (\mathcal{I}(E))\ (\boldsymbol{MI}\ [\![\ [\textbf{cond},\ \mathcal{I}(C_1),\ \mathcal{I}(C_2)]\ ]\!]\ (\boldsymbol{MC}\ [\![\ <>\ ]\!]\ (z)))$$

(axiom TC1)

$$= \boldsymbol{MC}\ (\mathcal{I}(E))\ (\boldsymbol{MI}\ ([\textbf{cond},\ \mathcal{I}(C_1),\ \mathcal{I}(C_2)])\ z)$$

(axiom I12)

$$\begin{aligned}
= \boldsymbol{MC}\ (\mathcal{I}(E))\ (\lambda(\text{stk},m,i,o).\ [&\boldsymbol{lg}(\text{stk}) < 1 \rightarrow \textbf{stack-underflow},\\
&(\text{is-bool}(\boldsymbol{hd}(\text{stk})) \rightarrow\\
&\quad(\boldsymbol{hd}(\text{stk}) \rightarrow \boldsymbol{MC}\ [\![\ \mathcal{I}(C_1)\ ]\!]\ (z)\ ((\boldsymbol{tl}(\text{stk}),m,i,o)),\\
&\quad\quad \boldsymbol{MC}\ [\![\ \mathcal{I}(C_2)\ ]\!]\ (z)\ ((\boldsymbol{tl}(\text{stk}),m,i,o))),\\
&\quad\textbf{non-bool-value})])
\end{aligned}$$

(MC-equals-ME lemma)

$$\begin{aligned}
= \boldsymbol{ME}\ (\mathcal{I}(E))\ (\lambda(\text{stk},m,i,o).\ [&\boldsymbol{lg}(\text{stk}) < 1 \rightarrow \textbf{stack-underflow},\\
&(\text{is-bool}(\boldsymbol{hd}(\text{stk})) \rightarrow\\
&\quad(\boldsymbol{hd}(\text{stk}) \rightarrow \boldsymbol{MC}\ [\![\ \mathcal{I}(C_1)\ ]\!]\ (z)\ ((\boldsymbol{tl}(\text{stk}),m,i,o)),\\
&\quad\quad \boldsymbol{MC}\ [\![\ \mathcal{I}(C_2)\ ]\!]\ (z)\ ((\boldsymbol{tl}(\text{stk}),m,i,o))),\\
&\quad\textbf{non-bool-value})])
\end{aligned}$$

(STACK-HAS-ONE lemma)

$$\begin{aligned}
= \boldsymbol{ME}\ (\mathcal{I}(E))\ (&\lambda(\text{stk},m,i,o).\\
&\text{is-bool}(\boldsymbol{hd}(\text{stk})) \rightarrow (\boldsymbol{hd}(\text{stk}) \rightarrow \boldsymbol{MC}\ [\![\ \mathcal{I}(C_1)\ ]\!]\ (z)\ ((\boldsymbol{tl}(\text{stk}),m,i,o)),\\
&\quad \boldsymbol{MC}\ [\![\ \mathcal{I}(C_2)\ ]\!]\ (z)\ ((\boldsymbol{tl}(\text{stk}),m,i,o))),\\
&\quad \textbf{non-bool-value})
\end{aligned}$$

Axiom (C4)

Translate Axiom into L_{target}

C ⟦ **while** E C ⟧ (c) $=_{cont}$
  E ⟦ E ⟧ (λv. [is-bool(v) → (v → *C* ⟦ C ⟧ (*C* ⟦ **while** E C ⟧ (c)), c),
  non-bool-value])

(translate axiom using interpretation, *I*)

λCc. [*MC* (C) (c)] ⟦ *I*(**while** E C) ⟧ *I*(c) $=_{mcont}$
  λEk. [*H* (E) (k)] ⟦ *I*(E) ⟧
  (*I*(λv(m,i,o). [is-bool(v) →
  (v → *C* ⟦ C ⟧ (*C* ⟦ **while** E C ⟧ (c)) ((m,i,o)), c ((m,i,o))),
  non-bool-value]))

(simplify)

MC ([**loop**, *I*(E), *I*(C)]) (z) $=_{mcont}$
  H(*I*(E))
  (λv(stk,m,i,o). [is-bool(v) →
  (v → *MC* ⟦ *I*(C) ⟧ (*MC* ⟦ **loop** *I*(E) *I*(C) ⟧ (z)) ((stk,m,i,o)), z ((stk,m,i,o))),
  non-bool-value])

(simplify)

MC ([**loop**, *I*(E), *I*(C)]) (z) $=_{mcont}$
  ME (*I*(E)) (
  λ(stk,m,i,o). is-bool(***hd***(stk)) →
  (***hd***(stk) → *MC* ⟦ *I*(C) ⟧ (*MC* ⟦ **loop** *I*(E) *I*(C) ⟧ (z)) ((***tl***(stk),m,i,o)),
  z ((***tl***(stk),m,i,o))),
  non-bool-value)

Proof in T_{target}

MC ([**loop**, *I*(E), *I*(C)]) (z)

(axiom TC2)

= *MI* ⟦ [**loop**, *I*(E), *I*(C)] ⟧ (*MC* ⟦ <> ⟧ (z))

(axiom TC1)

= *MI* ([**loop**, *I*(E), *I*(C)]) z

(axiom I13)

= λ(stk,m,i,o). ***ME*** ⟦ ***I***(E) ⟧ (λ(stk,m,i,o). [is-bool(***hd***(stk)) →
(***hd***(stk) →
MC ⟦ ***I***(C) ⟧ (***MI*** ⟦ [**loop**, ***I***(E), ***I***(C)] ⟧ (z)) ((***tl***(stk),m,i,o)),
z((***tl***(stk),m,i,o))),
non-bool-value])

(MI-equals-MC lemma)

(λ(stk,m,i,o). ***ME***(X)z = ***ME***(X)z)

= ***ME*** (***I***(E)) (
λ(stk,m,i,o). is-bool(***hd***(stk)) →
(***hd***(stk) → ***MC*** ⟦ ***I***(C) ⟧ (***MC*** ⟦ **loop** ***I***(E) ***I***(C) ⟧ (z)) ((***tl***(stk),m,i,o)),
z ((***tl***(stk),m,i,o))),
non-bool-value)

Axiom (C5)

Translate Axiom into L_{target}

$\boldsymbol{C} [\![C_1 ; C_2]\!] (c) =_{cont} \boldsymbol{C} [\![C_1]\!] (\boldsymbol{C} [\![C_2]\!] (c))$

(translate axiom using interpretation, $\boldsymbol{I}$)
(reduce λ-expressions)

$\boldsymbol{MC} (\boldsymbol{I}(C_1 ; C_2)) (\boldsymbol{I}(c)) =_{mcont} \boldsymbol{MC} (\boldsymbol{I}(C_1)) (\boldsymbol{I}(\boldsymbol{C} [\![C_2]\!] (c)))$

(translate axiom using interpretation, $\boldsymbol{I}$)

$\boldsymbol{MC} (\boldsymbol{I}(C_1) \bullet \boldsymbol{I}(C_2)) (z) =_{mcont} \boldsymbol{MC} (\boldsymbol{I}(C_1)) (\boldsymbol{MC} (\boldsymbol{I}(C_2)) (\boldsymbol{I}(c)))$

(translate axiom using interpretation, $\boldsymbol{I}$)

$\boldsymbol{MC} (\boldsymbol{I}(C_1) \bullet \boldsymbol{I}(C_2)) (z) =_{mcont} \boldsymbol{MC} (\boldsymbol{I}(C_1)) (\boldsymbol{MC} (\boldsymbol{I}(C_2)) (z))$

Proof in T_{target}

$\boldsymbol{MC} (\boldsymbol{I}(C_1) \bullet \boldsymbol{I}(C_2)) (z)$

(axiom TC3)

$= \boldsymbol{MC} (\boldsymbol{I}(C_1)) (\boldsymbol{MC} (\boldsymbol{I}(C_2)) (z))$

Axiom (P1)

Translate Axiom into L_{target}

$\mathcal{P}$ ⟦ **begin** P ⟧ (i) $=_{output \oplus error}$ λa. [a $\in$ error $\rightarrow$ a, $\mathbf{hd}(\mathbf{tl}(\mathbf{tl}(a)))$]
($\mathcal{C}$ ⟦ P ⟧ (c_o) (m_o,i,<>))

where $c_o = \lambda s.s$

$\forall I \in id,\ m_o(I) =$ **unbound**

<> = initial empty output

(translate axiom using interpretation, $\mathcal{I}$)

λP. [$\mathcal{MP}$ (P) z_o] ⟦ $\mathcal{I}$(**begin** P) ⟧ ($\mathcal{I}$(i)) = λa. [a $\in$ error $\rightarrow$ a, a]
(λCc. [$\mathcal{MC}$ (C) (c)] ⟦ $\mathcal{I}$(P) ⟧ $\mathcal{I}(c_o)$ ((<>,m_o,i,<>)))

(reduce λ-expressions)

$\mathcal{MP}$ ($\mathcal{I}$(**begin** P)) z_o $=_{mcont}$
λ(stk,m,i,o). ($\mathcal{MC}$ ($\mathcal{I}$(P)) ($\mathcal{I}(c_o)$) ((<>,m_o,i,<>)))

(translate axiom using interpretation, $\mathcal{I}$)

$\mathcal{MP}$ ⟦ **[start]** • $\mathcal{I}$(P) • **[halt]** ⟧ z_o $=_{mcont}$
λ(stk,m,i,o). ($\mathcal{MC}$ ($\mathcal{I}$(P)) (z_o) ((<>,m_o,i,<>)))

Proof in T_{target}

$\mathcal{MP}$ ⟦ **[start]** • $\mathcal{I}$(P) • **[halt]** ⟧ z_o

(axioms TC1, TC2 and TC3)

= $\mathcal{MI}$ ⟦ **[start]** ⟧ ($\mathcal{MP}$ ⟦ $\mathcal{I}$(P) ⟧ ($\mathcal{MI}$ ⟦ **[halt]** ⟧ z_o))

(axiom I14)

= λ(stk,m,i,o). ($\mathcal{MP}$ ⟦ $\mathcal{I}$(P) ⟧ ($\mathcal{MI}$ ⟦ **[halt]** ⟧ z_o)) ((<>,m_o,i,<>))

(axiom I15)

$= \lambda(stk,m,i,o).\ \boldsymbol{MP}\ [\![\ \boldsymbol{I}(P)\ ]\!]\ z_o\ ((<>,m_o,i,<>))$

(MP-equals-MC lemma)

$= \lambda(stk,m,i,o).\ \boldsymbol{MC}\ [\![\ \boldsymbol{I}(P)\ ]\!]\ z_o\ ((<>,m_o,i,<>))$

Lemmas

STACK-NOT-EMPTY Lemma

$\boldsymbol{ME}(\boldsymbol{I}(E))z$ = err, or
$\lambda(stk,m,i,o).\ z\ ((<v> \bullet stk,m,i',o))$
where err: error, E: exp

Proof

$E \in \{$**0**, **1**, **true**, **false**, **read**, I_i, **not** E_1, $E_1 = E_2$, $E_1 + E_2\}$
where E_1: exp and E_2: exp

Basis

$E \in \{$**0**, **1**, **true**, **false**, **read**, $I_i\}$

$\therefore \boldsymbol{I}(E) \in \{$[**loadn**, 0], [**loadn**, 1], [**loadb**,tt], [**loadb**, ff], [**read**], [**load**, I_i]$\}$

$\therefore \boldsymbol{ME}\ (\boldsymbol{I}(E))\ (z)$

(TE2 and TE1 axioms)

$= \boldsymbol{MI}\ (\boldsymbol{I}(E))\ z$

(definition of ***MI***)

=err, or
$\lambda(stk,m,i,o).\ z\ ((v \bullet stk,m,i',o))$
where $v \in \{0,\ 1,\ tt,\ ff,\ \boldsymbol{hd}(i),\ m(I_i)\}$
and $i' = \boldsymbol{tl}(i)$, if $\boldsymbol{I}(E)$=[**read**]
i, otherwise

Induction step, assume property true for all expression constituents of the expression

$E \in \{$**not** E_1, $E_1 = E_2$, $E_1 + E_2\}$

$\boldsymbol{ME}(\boldsymbol{I}(E_1) \bullet$ [**not**]) z

(TE3 and TE1 axioms)

= ***ME***(***I***(E$_1$)) (***MI***([**not**]) z)

(assumption)

= λ(stk,m,i,o). (***MI***([**not**]) z) ((<v> • stk,m,i′,o))

(definition of ***MI***)

= λ(stk,m,i,o). z ((<~v> • stk,m,i′,o)), or err

ME(***I***(E$_1$) • ***I***(E$_2$) • [**eq**]) z

(TE3 and TE1 axioms)

= ***ME***(***I***(E$_1$) • ***I***(E$_2$)) (***MI***([**eq**]) z)

(assumption)

= λ(stk,m,i,o). (***MI***([**eq**]) z) ((<v$_1$> • <v$_2$> • stk,m,i′,o))

(definition of ***MI***)

= λ(stk,m,i,o). z ((<v$_1$=v$_2$> • stk,m,i′,o)), or err

ME(***I***(E$_1$) • ***I***(E$_2$) • [**add**]) z

(TE3 and TE1 axioms)

= ***ME***(***I***(E$_1$) • ***I***(E$_2$)) (***MI***([**add**]) z)

(assumption)

= λ(stk,m,i,o). (***MI***([**add**]) z) ((<v$_1$> • <v$_2$> • stk,m,i′,o))

(definition of ***MI***)

= λ(stk,m,i,o). z ((<v$_1$+v$_2$> • stk,m,i′,o)), or err

STACK-HAS-ONE Lemma

$\mathcal{ME}$ ($\mathcal{I}$(E)) (λ(stk,m,i,o). $\mathbf{lg}$(stk) < 1 → A, B)
= $\mathcal{ME}$ [[$\mathcal{I}$(E)]] (λ(stk,m,i,o). B)
where A: mans, B: mans and E: exp

Proof

$\mathcal{ME}$ ($\mathcal{I}$(E)) (λ(stk,m,i,o). $\mathbf{lg}$(stk) < 1 → A, B)

(STACK-NOT-EMPTY lemma)

= err, or
λ(stk,m,i,o). (λ(stk,m,i,o). $\mathbf{lg}$(stk) < 1 → A, B)
((<v> • stk,m,i',o))

($\mathbf{lg}$(stk) ≥ 1)

= err, or
λ(stk,m,i,o). (λ(stk,m,i,o). B) ((<v> • stk,m,i',o))

= $\mathcal{ME}$ ($\mathcal{I}$(E)) (λ(stk,m,i,o). B)

STACK-HAS-TWO Lemma

$\boldsymbol{ME}$ ⟦ $\boldsymbol{I}(E_1)$ ⟧ ($\boldsymbol{ME}$ ⟦ $\boldsymbol{I}(E_2)$ ⟧ (λ(stk,m,i,o). **lg**(stk) < 2 → A, B))
= $\boldsymbol{ME}$ ⟦ $\boldsymbol{I}(E_1)$ ⟧ ($\boldsymbol{ME}$ ⟦ $\boldsymbol{I}(E_2)$ ⟧ (λ(stk,m,i,o). B))
where A: mans, B: mans, E_1: exp, and E_2: exp

Proof

$\boldsymbol{ME}$ ⟦ $\boldsymbol{I}(E_1)$ ⟧ ($\boldsymbol{ME}$ ⟦ $\boldsymbol{I}(E_2)$ ⟧ (λ(stk,m,i,o). **lg**(stk) < 2 → A, B))

(STACK-NOT-EMPTY lemma)

= err, or
λ(stk,m,i,o). $\boldsymbol{ME}$ ⟦ $\boldsymbol{I}(E_2)$ ⟧ (λ(stk,m,i,o). **lg**(stk) < 2 → A, B))
((<v> • stk,m,i,o))

(STACK-NOT-EMPTY lemma)

= err, or
λ(stk,m,i,o). [λ(stk′,m′,i′,o′). (λ(stk″,m″,i″,o″). **lg**(stk″) < 2 → A, B)
((<u> • stk′,m′,i′,o′))] ((<v> • stk,m,i,o))

(reduce λ-expression)

= err, or
λ(stk,m,i,o). (λ(stk″,m″,i″,o″). **lg**(stk″) < 2 → A, B)
((<u> • <v> • stk,m,i,o))

(**lg**(stk) ≥ 2)

= err, or
λ(stk,m,i,o). (λ(stk″,m″,i″,o″). B)
((<u> • <v> • stk,m,i,o))

= $\boldsymbol{ME}$ ⟦ $\boldsymbol{I}(E_1)$ ⟧ ($\boldsymbol{ME}$ ⟦ $\boldsymbol{I}(E_2)$ ⟧ (λ(stk,m,i,o). B))

MC-equals-ME Lemma

> $\mathcal{MC}(\mathcal{I}(E)) = \mathcal{ME}(\mathcal{I}(E))$, where E: exp
>
> **Proof**
>
> $\mathcal{MC}$: code $\rightarrow$ mcont $\rightarrow$ mcont
> $\mathcal{ME}$: ecode $\rightarrow$ mcont $\rightarrow$ mcont
> code = instr* = ecode
> $\mathcal{I}$(E): ecode
> ∴ $\mathcal{MC}(\mathcal{I}(E)) = \mathcal{ME}(\mathcal{I}(E))$

MP-equals-MC Lemma

> $\mathcal{MP}(\mathcal{I}(P)) = \mathcal{MC}(\mathcal{I}(P))$, where P: com
>
> **Proof**
>
> $\mathcal{MC}$: code $\rightarrow$ mcont $\rightarrow$ mcont
> $\mathcal{MP}$: pcode $\rightarrow$ mcont $\rightarrow$ mcont
> code = instr* = pcode
> $\mathcal{I}$(P): code
> ∴ $\mathcal{MP}(\mathcal{I}(P)) = \mathcal{MC}(\mathcal{I}(P))$

MI-equals-MC Lemma

> $\mathcal{MC}(X) = \mathcal{MI}(X)$, where X: instr
>
> **Proof**
>
> $\mathcal{MC}$: code $\rightarrow$ mcont $\rightarrow$ mcont
> $\mathcal{MI}$: instr $\rightarrow$ mcont $\rightarrow$ mcont
> code = instr*
> ∴ by axioms TC1 and TC2,
> $\mathcal{MC}(X) = \mathcal{MI}(X)$

Appendix E

Specification of CS-Tiny2

Theory for Source Language, T_{source}

Language for Source Language, L_{source}

Language Elements	Defined Language	Defining Language
domains	id = {**I**, $\mathbf{I_1}$, $\mathbf{I_2}$...} exp com prog	num bool value = num ⊕ bool input = value* output = value* mem = id → [value ⊕ {**unbound**}] state = mem ⊗ input ⊗ output error = {**empty-input**, **unbound-var**, **non-bool-value**, **non-num-value** **empty-stk-error**} ans = state ⊕ error cont = state → ans econt = value → cont $econt_0$ = cont $econt_{n+1}$ = value → $econt_n$

function symbols	**0**: $\rightarrow$ exp **1**: $\rightarrow$ exp **true**: $\rightarrow$ exp **false**: $\rightarrow$ exp **read**: $\rightarrow$ exp $\{I, I_1, I_2 \ldots\}$: $\rightarrow$ exp **not**: exp $\rightarrow$ exp **=**: exp $\otimes$ exp $\rightarrow$ exp **+**: exp $\otimes$ exp $\rightarrow$ exp **:=**: id $\otimes$ exp $\rightarrow$ com **output**: exp $\rightarrow$ com **if**: exp $\otimes$ com $\otimes$ com $\rightarrow$ com **while**: exp $\otimes$ com $\rightarrow$ com ;: com $\otimes$ com $\rightarrow$ com **begin**: com $\rightarrow$ prog	***E***: exp $\rightarrow$ econt $\rightarrow$ econt ***C***: com $\rightarrow$ cont $\rightarrow$ cont ***P***: prog $\rightarrow$ input $\rightarrow$ [output $\oplus$ error] ***hd***: value* $\rightarrow$ value $\oplus$ error ***tl***: value* $\rightarrow$ value* _ • _: value $\otimes$ value* $\rightarrow$ value* _+_: num $\otimes$ num $\rightarrow$ num
predicate symbols		null: value* $\rightarrow$ bool $\in$ error: ans $\rightarrow$ bool
individual variable symbols		k: econt v, v′, v_i: value, $1 \le i \le n$ m: mem i: input o: output C, C_1, C_2: com E, E_1, E_2: exp c_o, c: cont a_1, a_2: ans I, I′: id v*: value* s: state P: prog (m,i,o): state b: bool

Axioms for Source Language, A_{source}

(E1a) $\boldsymbol{E}\, [\![\, \mathbf{0} \,]\!]\, (k) =_{econt} k(0)$

(E1b) $\boldsymbol{E}\, [\![\, \mathbf{1} \,]\!]\, (k) =_{econt} k(1)$

(E2a)	$\boldsymbol{E}[\![\ \textbf{true}\ ]\!]\ (k) =_{econt} k(tt)$
(E2b)	$\boldsymbol{E}[\![\ \textbf{false}\ ]\!]\ (k) =_{econt} k(ff)$
(E3)	$\boldsymbol{E}[\![\ \textbf{read}\ ]\!]\ (k) =_{econt}$ $\lambda v_1 \ldots v_n (m,i,o).\ [null(i) \rightarrow \textbf{empty-input},$ $k(\boldsymbol{hd}(i))(v_1)\ldots(v_n)((m,\boldsymbol{tl}(i),o))]$
(E4)	$\boldsymbol{E}[\![\ I\ ]\!]\ (k) =_{econt}$ $\lambda v_1 \ldots v_n (m,i,o).\ [m(I) = \textbf{unbound} \rightarrow \textbf{unbound-var},$ $k(m(I))(v_1)\ldots(v_n)((m,i,o))]$
(E5)	$\boldsymbol{E}[\![\ \textbf{not}\ E\ ]\!]\ (k) =_{econt}$ $\boldsymbol{E}[\![\ E\ ]\!]\ (\lambda v v_1 \ldots v_n s.\ [is\text{-}bool(v) \rightarrow k(\sim v)(v_1)\ldots(v_n)(s),$ $\textbf{non-bool-value}])$
(E6)	$\boldsymbol{E}[\![\ E_1 = E_2\ ]\!]\ (k) =_{econt} \boldsymbol{E}[\![\ E_1\ ]\!]\ (\ \boldsymbol{E}[\![\ E_2\ ]\!]\ (\ \lambda v' v.\ [k(v =_{value} v')]))$
(E7)	$\boldsymbol{E}[\![\ E_1 + E_2\ ]\!]\ (k) =_{econt}$ $\boldsymbol{E}[\![\ E_1\ ]\!]\ (\ \boldsymbol{E}[\![\ E_2\ ]\!]\ (\ \lambda v' v.\ [is\text{-}num(v)\ \&\ is\text{-}num(v') \rightarrow k(v+v'),$ $\textbf{non-num-value}]))$
(C1)	$\boldsymbol{C}[\![\ I := E\ ]\!]\ (c) =_{cont} \boldsymbol{E}[\![\ E\ ]\!]\ (\lambda v(m,i,o).\ [c((m[v/I],i,o))])$
(C2)	$\boldsymbol{C}[\![\ \textbf{output}\ E\ ]\!]\ (c) =_{cont} \boldsymbol{E}[\![\ E\ ]\!]\ (\lambda v(m,i,o).\ [c((m,i,o \bullet \langle v \rangle))])$
(C3)	$\boldsymbol{C}[\![\ \textbf{if}\ E\ C_1\ C_2\ ]\!]\ (c) =_{cont}$ $\boldsymbol{E}[\![\ E\ ]\!]\ (\lambda v.\ [is\text{-}bool(v) \rightarrow (v \rightarrow \boldsymbol{C}[\![\ C_1\ ]\!]\ (c),\ \boldsymbol{C}[\![\ C_2\ ]\!]\ (c)),$ $\textbf{non-bool-value}])$
(C4)	$\boldsymbol{C}[\![\ \textbf{while}\ E\ C\ ]\!]\ (c) =_{cont}$ $\boldsymbol{E}[\![\ E\ ]\!]\ (\lambda v.\ [is\text{-}bool(v) \rightarrow (v \rightarrow \boldsymbol{C}[\![\ C\ ]\!]\ (\boldsymbol{C}[\![\ \textbf{while}\ E\ C\ ]\!]\ (c)),\ c),$ $\textbf{non-bool-value}]$
(C5)	$\boldsymbol{C}[\![\ C_1 ; C_2\ ]\!]\ (c) =_{cont} \boldsymbol{C}[\![\ C_1\ ]\!]\ (\ \boldsymbol{C}[\![\ C_2\ ]\!]\ (c)\ )$
(P1)	$\boldsymbol{P}[\![\ \textbf{begin}\ P\ ]\!]\ (i) =_{output} \lambda a.\ [a \in error \rightarrow a,\ \boldsymbol{hd}(\boldsymbol{tl}(\boldsymbol{tl}(a)))]$ $(\ \boldsymbol{C}[\![\ P\ ]\!]\ (c_o)\ (m_o,i,\langle\rangle))$ where $c_o = \lambda s.s$ $\forall I \in id,\ m_o(I) = \textbf{unbound}$ $\langle\rangle$ = initially empty output
(A1)	$m[v/I](I') =_{value \oplus \{\textbf{unbound}\}} (I =_{id} I' \rightarrow v,\ m(I'))$
(A2a)	$\boldsymbol{hd}(\langle\rangle) =_{error \oplus value} \textbf{empty-stk-error}$
(A2b)	$\boldsymbol{hd}(\langle v \rangle) =_{error \oplus value} v$
(A2c)	$\boldsymbol{hd}(\langle v \rangle \bullet v^*) =_{error \oplus value} v$

(A2d) $\boldsymbol{tl}(\langle v \rangle \bullet v^*) = v^*$

(A2e) $\boldsymbol{hd}(v^*) \bullet \boldsymbol{tl}(v^*) = v^*$

Appendix F

Implementation of CS-Small

Theory for Source Language, T_{source}

Language for Source Language, L_{source}

Language Elements	Defined Language	Defining Language
domains	id = {**I**, $\mathbf{I_1}$, $\mathbf{I_2}$...}	num
	exp	bool
	com	loc = {input, l_1, ... , l_n}
	prog	bv
	decl	rv = bool ⊕ bv
	bas	dv = loc ⊕ rv ⊕ proc
	opr	sv = file ⊕ rv
		file = rv^*
		env = id → (dv ⊕ {**unbound**})
		store = loc → (sv ⊕ {**unused**})
		state = env ⊗ store ⊗ ans
		ans = {error, stop} ⊕ (rv ⊗ ans)
		cont = state → state
		econt = dv → cont
		dcont = env → cont
		proc = cont → econt

function symbols	**B**: → bas	***E***: exp → econt → cont
	O: → opr	***C***: com → cont → cont
	true: → exp	***P***: prog → [file → ans]
	false: → exp	***hd***: rv* → rv ⊕ {error}
	read: → exp	***tl***: rv* → rv*
	{**I**, $\mathbf{I_1}$, $\mathbf{I_2}$...}: → exp	_ • _: rv ⊗ rv* → rv*
	B: exp	_+_: num ⊗ num → num
	_ (_): exp ⊗ exp → exp	***R***: exp → econt → cont
	O: exp ⊗ exp → exp	***D***: decl → dcont → cont
	:=: exp ⊗ exp → com	***B***: bas → bv
	output: exp → com	***O***: opr → (rv ⊗ rv) → econt → cont
	if: exp ⊗ com ⊗ com → com	
	while: exp ⊗ com → com	
	;: com ⊗ com → com	
	begin: decl ⊗ com → com	
	program: com → prog	
	const: id ⊗ exp → decl	
	var: id ⊗ exp → decl	
	proc: id ⊗ id ⊗ com → decl	
	,: decl ⊗ decl → decl	
predicate symbols		null: rv* → bool
individual variable symbols		k: econt
		C, C_1, C_2: com
		E, E_1, E_2: exp
		c_o, c: cont
		P: prog
		b: bool
		n: num
		l: loc
		e: bv
		D, D_1, D_2: decl
		d: dv
		v: sv
		e: rv
		i: file
		r: env
		s: store
		u: dcont
		p: proc
		a: ans

Axioms for Source Language, A_{source}

(E1) $\boldsymbol{E} \llbracket \mathbf{B} \rrbracket (k) =_{cont} k(\boldsymbol{B} \llbracket \mathbf{B} \rrbracket)$

(E2a) $\boldsymbol{E} \llbracket \mathbf{true} \rrbracket (k) =_{cont} k(\mathrm{TRUE})$

(E2b) $\boldsymbol{E} \llbracket \mathbf{false} \rrbracket (k) =_{cont} k(\mathrm{FALSE})$

(E3) $\boldsymbol{E} \llbracket \mathbf{read} \rrbracket (k) =_{cont} \lambda(r,s,a).\ [\mathrm{null}(s(\mathrm{input})) \rightarrow \langle r, s, \langle a, \mathrm{error}\rangle\rangle,$
$k\ (\boldsymbol{hd}(s(\mathrm{input})))\ (\langle r, s[\boldsymbol{tl}(s(\mathrm{input}))/\mathrm{input}], a\rangle)]$

(E4) $\boldsymbol{E} \llbracket \mathrm{I} \rrbracket (k) =_{cont} \lambda(r,s,a).\ [r(\mathrm{I}) = \mathbf{unbound} \rightarrow \langle r, s, \langle a, \mathrm{error}\rangle\rangle,$
$k(r(\mathrm{I}))((r,s,a))]$

(E7) $\boldsymbol{E} \llbracket E_1\ \mathbf{O}\ E_2 \rrbracket (k) =_{cont} \boldsymbol{R} \llbracket E_1 \rrbracket (\lambda e'.\ \boldsymbol{R} \llbracket E_2 \rrbracket (\ \lambda e.\ [\boldsymbol{O} \llbracket \mathbf{O} \rrbracket (e', e)\ (k)]))$

(C1) $\boldsymbol{C} \llbracket E_1 := E_2 \rrbracket (c) =_{cont} \boldsymbol{E} \llbracket E_1 \rrbracket (\mathrm{loc?}\ (\lambda l.\ \boldsymbol{R} \llbracket E_2 \rrbracket\ (\mathrm{update}(l)(c)))$

(C2) $\boldsymbol{C} \llbracket \mathbf{output}\ E \rrbracket (c) =_{cont} \boldsymbol{R} \llbracket E \rrbracket (\lambda e(r,s,a).\ [c((r, s, \langle a, e\rangle))])$

(C3) $\boldsymbol{C} \llbracket E_1(E_2) \rrbracket (c) =_{cont} \boldsymbol{E} \llbracket E_1 \rrbracket (\mathrm{proc?}\ (\lambda p.\ \boldsymbol{E} \llbracket E_2 \rrbracket\ (p(c))))$

(C4) $\boldsymbol{C} \llbracket \mathbf{if}\ E\ C_1\ C_2 \rrbracket (c) =_{cont} \boldsymbol{R} \llbracket E \rrbracket (\mathrm{bool?}\ (\lambda e.\ e \rightarrow (\boldsymbol{C} \llbracket C_1 \rrbracket (c),$
$\boldsymbol{C} \llbracket C_2 \rrbracket (c))))$

(C5) $\boldsymbol{C} \llbracket \mathbf{while}\ E\ C \rrbracket (c) =_{cont}$
$\boldsymbol{R} \llbracket E \rrbracket (\mathrm{bool?}\ (\lambda e.\ e \rightarrow (\ \boldsymbol{C} \llbracket C \rrbracket\ (\boldsymbol{C} \llbracket \mathbf{while}\ E\ C \rrbracket (c)), c))$

(C6) $\boldsymbol{C} \llbracket \mathbf{begin}\ D\ C \rrbracket (c) =_{cont}$
$\lambda(r, s, a).\ \boldsymbol{D} \llbracket D \rrbracket (\lambda r'.\ \boldsymbol{C} \llbracket C \rrbracket (c)\ (r[r'], s, a))\ (r, s, a)$

(C7) $\boldsymbol{C} \llbracket C_1 ; C_2 \rrbracket (c) =_{cont} \boldsymbol{C} \llbracket C_1 \rrbracket (\ \boldsymbol{C} \llbracket C_2 \rrbracket (c)\ )$

(P) $\boldsymbol{P} \llbracket \mathbf{program}\ C \rrbracket (i) =_{ans}$
$\boldsymbol{C} \llbracket C \rrbracket (\lambda(r, s, a).\ \langle r, s, \langle a, \mathrm{stop}\rangle\rangle)(\langle r_0, s_0[i/\mathrm{input}], a_0\rangle)$
where $\forall l \in \mathrm{loc},\ s_0(l) = \mathbf{unused}$
$\forall \mathrm{I} \in \mathrm{id},\ r_0(\mathrm{I}) = \mathbf{unbound}$
a_0 = initially empty output

(R) $\boldsymbol{R} \llbracket E \rrbracket (k) =_{cont} \boldsymbol{E} \llbracket E \rrbracket (\mathrm{deref}\ (\mathrm{rv?}\ (k)))$

(D1) $\boldsymbol{D} \llbracket \mathbf{const}\ \mathrm{I}\ E \rrbracket (u) =_{cont} \boldsymbol{R} \llbracket E \rrbracket (\lambda e.\ u[e/\mathrm{I}])$

(D2) $\boldsymbol{D} \llbracket \mathbf{var}\ \mathrm{I}\ E \rrbracket (u) =_{cont} \boldsymbol{R} \llbracket E \rrbracket (\mathrm{ref}\ (\lambda i.\ u[i/\mathrm{I}]))$

(D3) $\boldsymbol{D} \llbracket \mathbf{proc}\ \mathrm{I}\ \mathrm{I}_1\ C \rrbracket (u) =_{cont}$
$\lambda(r, s, a).\ \ u[(\lambda ce(r', s', a').\ \ \boldsymbol{C}(C)(c)(\langle r[e/\mathrm{I}_1], s', a'\rangle))/\mathrm{I}](\langle r, s, a\rangle)$

(D5) $\boldsymbol{D} \llbracket D_1, D_2 \rrbracket (u) =_{cont}$
$\lambda(r, s, a).\ \boldsymbol{D} \llbracket D_1 \rrbracket (\lambda r_1.\ \boldsymbol{D} \llbracket D_2 \rrbracket (\lambda r_2.\ u[r_1[r_2]])\ (r[r_1], s, a))\ (\langle r, s, a\rangle)$

(A1) $r[e/I](I') =_{dv \oplus \{\textbf{unbound}\}} (I =_{id} I' \rightarrow e,\ r(I'))$

(A2a) $\boldsymbol{hd}(\langle\rangle) =_{\{error\} \oplus rv} error$

(A2b) $\boldsymbol{hd}(\langle e\rangle) =_{\{error\} \oplus rv} e$

(A2c) $\boldsymbol{hd}(\langle e\rangle \bullet e^*) =_{\{error\} \oplus rv} e$

(A2d) $\boldsymbol{tl}(\langle e\rangle \bullet e^*) = e^*$

(A2e) $\boldsymbol{hd}(e^*) \bullet \boldsymbol{tl}(e^*) = e^*$

Abbreviations

$loc?: econt \rightarrow econt$
$loc? = \lambda ke.\ isloc(e) \rightarrow k(e),\ (\lambda(r, s, a).\ \langle r, s, \langle a, error\rangle\rangle)$

$proc?: econt \rightarrow econt$
$proc? = \lambda ke.\ isproc(e) \rightarrow k(e),\ (\lambda(r, s, a).\ \langle r, s, \langle a, error\rangle\rangle)$

$rv?: econt \rightarrow econt$
$rv? = \lambda ke.\ isrv(e) \rightarrow k(e),\ (\lambda(r, s, a).\ \langle r, s, \langle a, error\rangle\rangle)$

$bool?: econt \rightarrow econt$
$bool? = \lambda ke.\ isrv(e) \rightarrow (isbool(e) \rightarrow k(e),\ (\lambda(r, s, a).\ \langle r, s, \langle a, error\rangle\rangle)),$
$\quad (\lambda(r, s, a).\ \langle r, s, \langle a, error\rangle\rangle)$

$update: loc \rightarrow cont \rightarrow econt$
$update = \lambda lce(r, s, a).\ issv(e) \rightarrow c(\langle r, s[e/l], a\rangle),\ \langle r, s, \langle a, error\rangle\rangle$

$new: store \rightarrow (loc \oplus \{error\})$
$new = \lambda s.\ s(l_1) = \textbf{unused} \rightarrow l_1, \ldots, s(l_n) = \textbf{unused} \rightarrow l_n, error$

$ref: econt \rightarrow econt$
$ref = \lambda ke(r, s, a).\ new(s) = error \rightarrow \langle r, s, \langle a, error\rangle\rangle,$
$\quad update\ (new(s))\ (k(new(s)))\ (e)\ (\langle r, s, a\rangle)$

$deref: econt \rightarrow econt$
$deref = \lambda ke(r, s, a).\ isloc(e) \rightarrow (s(e) = \textbf{unused} \rightarrow \langle r, s, \langle a, error\rangle\rangle,\ k(s(e))(\langle r, s, a\rangle)),$
$\quad k(e)(\langle r, s, a\rangle)$

Theory for Target Language, T_{target}

Language for Target Language, L_{target}

Language Elements	Defined Language	Defining Language
domains	id	
	instr	loc = {input, l_1, ... , l_n}
	code = instr*	bv
	ecode = instr*	rv = bool ⊕ bv
	num	mdv = loc ⊕ rv ⊕ mproc
	bool	sv = file ⊕ rv
	pcode = instr*	file = rv*
	ocode = instr*	alist = (id ⊗ mdv)*
	dcode = instr*	menv = {<>} ⊕ (alist ⊗ menv)
	bas	store = loc → (sv ⊕ {**unused**})
		mstate = menv ⊗ store ⊗ ans ⊗ stack ⊗ dump
		ans = {error, stop} ⊕ (rv ⊗ ans)
		mproc = mcont → mcont
		mcont = mstate → mstate
		stack = mdv*
		dump = menv*
function symbols	**start**: → instr	***MI***: instr → mcont → mcont
	halt: → instr	***MC***: code → mcont → mcont
	loadv: bas → instr	***ME***: ecode → mcont → mcont
	loadb: bool → instr	***MP***: pcode → mcont → mcont
	read: → instr	***hd***: D* → D ⊕ {error} where D is either mdv, menv, or (id ⊗ mdv)
	load: id → instr	***tl***: D* → D*
	pcall: → instr	***lg***: D* → num
	mkproc: dcode → instr	_ • _: D ⊗ D* → D*
	ret: → instr	_+_: num ⊗ num → num
	store: → instr	***O***: ocode → mcont → mcont
	output: → instr	***B***: bas → bv
	cond: code ⊗ code → instr	***MD***: dcode → mcont → mcont
	loop: ecode ⊗ code → instr	
	init: → instr	
	bind: id → instr	
	begin: → instr	
	end: → instr	
	deref: → instr	
	op: → instr	
	start: → instr	

predicate symbols	null: $D^* \to$ bool <: num $\otimes$ num $\to$ bool
individual variable symbols	z: mcont stk: stack (r, s, a, stk, d): mstate ID: id P, Q: code T: ecode I: instr

Axioms for Target Language, A_{target}

(I1) ***MI*** ⟦ [**loadv, B**] ⟧ (z) ((r, s, a, stk, d)) $=_{mstate}$ z((r, s, a, <***B*(B)** • stk>, d))

(I3a) ***MI*** ⟦ [**loadb**, TRUE] ⟧ (z) ((r, s, a, stk, d)) $=_{mstate}$ z((r, s, a, <TRUE • stk>, d))

(I3b) ***MI*** ⟦ [**loadb**, FALSE] ⟧ (z) ((r, s, a, stk, d)) $=_{mstate}$ z((r, s, a, <FALSE • stk>, d))

(I5) ***MI*** ⟦ [**read**] ⟧ (z) ((r, s, a, stk, d)) $=_{mstate}$
[null(s(input)) → <r, s, <a, error>, stk, d>,
z((r, s[***tl***(s(input))/input], a, <***hd***(s(input)) • stk>, d))]

(I6) ***MI*** ⟦ [**load**,ID] ⟧ (z) ((r, s, a, stk, d)) $=_{mstate}$
[dv?(ID)(r) = **unbound** → <r, s, <a, error>, stk, d>,
z((r, s, a, <dv?(ID)(r) • stk>, d))]
where dv?(ID)(r) = (null(r) → **unbound**,
let v = search(ID)(***hd***(r)) in (v = **unbound** → dv?(ID)(***tl***(r)), v))
and search(ID)(r) = (null(r) → **unbound**,
pr1(***hd***(r)) = ID → pr2(***hd***(r)), search(ID)(***tl***(r)))

(I8) ***MI*** ⟦ [**op**] ⟧ (z) ((r, s, a, stk, d)) $=_{mstate}$
[***lg***(stk) < 2 → <r, s, <a, error>, stk, d>,
z((r, s, a, <***O*(op)**(***hd***(stk), ***hd***(***tl***(stk)))(z) • ***tl***(***tl***(stk))> d))]

(I10) ***MI*** ⟦ [**store**] ⟧ (z) ((r, s, a, stk, d)) $=_{mstate}$
lg(stk) < 1 → <r, s, <a, error>, stk, d>,
isloc(***hd***(***tl***(stk))) →
[issv(***hd***(stk)) → z(<r,s[***hd***(stk)/***hd***(***tl***(stk))], a, ***tl***(***tl***(stk)), d>),
<r, s, <a, error>, stk, d>],
<r, s, <a, error>, stk, d>

(I11) $$\boldsymbol{MI}\ [\![\ [\mathbf{output}]\ ]\!]\ (z)\ ((r, s, a, stk, d)) =_{mstate}$$
$$[\boldsymbol{lg}(stk) < 1 \rightarrow \langle r, s, \langle a, error\rangle, stk, d\rangle,$$
$$z((r, s, \langle a, \boldsymbol{hd}(stk)\rangle, \boldsymbol{tl}(stk), d))]$$

(I12) $$\boldsymbol{MI}\ [\![\ [\mathbf{cond}, P, Q]\ ]\!]\ (z)\ ((r, s, a, stk, d)) =_{mstate}$$
$$[\boldsymbol{lg}(stk) < 1 \rightarrow \langle r, s, \langle a, error\rangle, stk, d\rangle,$$
$$(is\text{-}bool(\boldsymbol{hd}(stk)) \rightarrow$$
$$(\boldsymbol{hd}(stk) \rightarrow \boldsymbol{MC}\ [\![\ P\ ]\!]\ (z)\ ((r, s, a, \boldsymbol{tl}(stk), d)),$$
$$\boldsymbol{MC}\ [\![\ Q\ ]\!]\ (z)\ ((r, s, a, \boldsymbol{tl}(stk), d))),$$
$$\langle r, s, \langle a, error\rangle, stk, d\rangle)]$$

(I13) $$\boldsymbol{MI}\ [\![\ [\mathbf{loop}, T, P]\ ]\!]\ (z)\ ((r, s, a, stk, d)) =_{mstate}$$
$$\boldsymbol{ME}\ [\![\ T\ ]\!]\ (\lambda(r, s, a, stk, d).\ [is\text{-}bool(\boldsymbol{hd}(stk)) \rightarrow$$
$$(\boldsymbol{hd}(stk) \rightarrow$$
$$\boldsymbol{MC}\ [\![\ P\ ]\!]\ (\boldsymbol{MI}\ [\![\ [\mathbf{loop}, T, P]\ ]\!]\ (z))\ ((r, s, a, \boldsymbol{tl}(stk), d)),$$
$$z((r, s, a, \boldsymbol{tl}(stk), d))),$$
$$\langle r, s, \langle a, error\rangle, stk, d\rangle])((r, s, a, stk, d))$$

(I14) $$\boldsymbol{MI}\ [\![\ [\mathbf{start}]\ ]\!]\ (z)\ ((r, s, a, stk, d)) =_{mstate} z((\langle\rangle, s_o, a_o, \langle\rangle, \langle\rangle))$$

(I15) $$\boldsymbol{MI}\ [\![\ [\mathbf{halt}]\ ]\!]\ (z)\ ((r, s, a, stk, d)) =_{mstate} (r, s, \langle a, stop\rangle, stk, d)$$

(I16) $$\boldsymbol{MI}\ [\![\ [\mathbf{deref}]\ ]\!]\ (z)\ ((r, s, a, stk, d)) =_{mstate}$$
$$isloc(\boldsymbol{hd}(stk)) \rightarrow$$
$$[s(\boldsymbol{hd}(stk)) = \mathbf{unused} \rightarrow (r, s, \langle a, error\rangle, stk, d),$$
$$(isrv(s(\boldsymbol{hd}(stk))) \rightarrow z(\langle r, s, a, \langle s(\boldsymbol{hd}(stk)) \bullet \boldsymbol{tl}(stk)\rangle, d\rangle),$$
$$\langle r, s, \langle a, error\rangle, stk, d\rangle)],$$
$$[isrv(\boldsymbol{hd}(stk)) \rightarrow z(\langle r, s, a, stk, d\rangle), \langle r, s, \langle a, error\rangle, stk, d\rangle]$$

(I17) $$\boldsymbol{MI}\ [\![\ [\mathbf{begin}]\ ]\!]\ (z)\ ((r, s, a, stk, d)) =_{mstate} z(\langle\langle\langle\rangle, r\rangle, s, a, stk, d\rangle)$$

(I18) $$\boldsymbol{MI}\ [\![\ [\mathbf{end}]\ ]\!]\ (z)\ ((r, s, a, stk, d)) =_{mstate} z(\langle \boldsymbol{tl}(r), s, a, stk, d\rangle)$$

(I19) $$\boldsymbol{MI}\ [\![\ [\mathbf{bind}\ ID]\ ]\!]\ (z)\ ((r, s, a, stk, d)) =_{mstate}$$
$$\boldsymbol{lg}(stk) < 1 \rightarrow \langle r, s, \langle a, error\rangle, stk, d\rangle,$$
$$z(\langle\langle\langle ID, \boldsymbol{hd}(stk)\rangle \bullet pr1(r), pr2(r)\rangle, s, a, \boldsymbol{tl}(stk), d\rangle)$$

(I20) $$\boldsymbol{MI}\ [\![\ [\mathbf{init}]\ ]\!]\ (z)\ ((r, s, a, stk, d)) =_{mstate}$$
$$isloc(new(s)) \rightarrow z(\langle r, s[\boldsymbol{hd}(stk)/new(s)], a, \langle new(s) \bullet \boldsymbol{tl}(stk)\rangle, d\rangle),$$
$$\langle r, s, \langle a, error\rangle, stk, d\rangle$$

(I21) $$\boldsymbol{MI}\ [\![\ [\mathbf{mkproc}\ P]\ ]\!]\ (z)\ ((r, s, a, stk, d)) =_{mstate}$$
$$z(\langle r, s, a, \langle(\lambda z'(r', s', a', stk', d').\ \boldsymbol{MP}(P)(z')(r, s', a', stk', \langle r', d'\rangle)) \bullet stk\rangle, d\rangle)$$

(I22) $\boldsymbol{MI}\ [\![\ \mathbf{[pcall]}\ ]\!]\ (z)\ ((r, s, a, stk, d)) =_{mstate}$

$\boldsymbol{lg}(stk) < 2 \rightarrow <r, s, <a, error>, stk, d>,$

$\quad isproc(\boldsymbol{hd}(\boldsymbol{tl}(stk))) \rightarrow$

$\qquad (\boldsymbol{hd}(\boldsymbol{tl}(stk)))\ (z)\ (<r, s, a, <\boldsymbol{hd}(stk) \bullet \boldsymbol{tl}(\boldsymbol{tl}(stk))>, d>),$

$\qquad <r, s, <a, error>, stk, d>$

(I23) $\boldsymbol{MI}\ [\![\ \mathbf{[ret]}\ ]\!]\ (z)\ ((r, s, a, stk, d)) =_{mstate} z(<\boldsymbol{hd}(d), s, a, stk, \boldsymbol{tl}(d)>)$

(TC1) $\boldsymbol{MC}\ [\![\ <>\ ]\!]\ (z) =_{mcont} z$

(TC2) $\boldsymbol{MC}\ [\![\ <I>\bullet P\ ]\!]\ (z) =_{mcont} \boldsymbol{MI}\ [\![\ I\ ]\!]\ (\boldsymbol{MC}\ [\![\ P\ ]\!]\ (z))$

(TC3) $\boldsymbol{MC}\ [\![\ P\bullet Q\ ]\!]\ (z) =_{mcont} \boldsymbol{MC}\ [\![\ P\ ]\!]\ (\boldsymbol{MC}\ [\![\ Q\ ]\!]\ (z))$

(TE1) $\boldsymbol{ME}\ [\![\ <>\ ]\!]\ (z) =_{mcont} z$

(TE2) $\boldsymbol{ME}\ [\![\ <I>\bullet P\ ]\!]\ (z) =_{mcont} \boldsymbol{MI}\ [\![\ I\ ]\!]\ (\boldsymbol{ME}\ [\![\ P\ ]\!]\ (z))$

(TE3) $\boldsymbol{ME}\ [\![\ P\bullet Q\ ]\!]\ (z) =_{mcont} \boldsymbol{ME}\ [\![\ P\ ]\!]\ (\boldsymbol{ME}\ [\![\ Q\ ]\!]\ (z))$

(TP1) $\boldsymbol{MP}\ [\![\ <>\ ]\!]\ (z) =_{mcont} z$

(TP2) $\boldsymbol{MP}\ [\![\ <I>\bullet P\ ]\!]\ (z) =_{mcont} \boldsymbol{MI}\ [\![\ I\ ]\!]\ (\boldsymbol{MP}\ [\![\ P\ ]\!]\ (z))$

(TP3) $\boldsymbol{MP}\ [\![\ P\bullet Q\ ]\!]\ (z) =_{mcont} \boldsymbol{MP}\ [\![\ P\ ]\!]\ (\boldsymbol{MP}\ [\![\ Q\ ]\!]\ (z))$

(A1) $s[v/I](I') =_{sv \oplus \{\mathbf{unused}\}} (I =_{loc} I' \rightarrow v, s(I'))$

(A2a) $\boldsymbol{hd}(<>) =_{\{error\} \oplus D} error$

(A2b) $\boldsymbol{hd}(<v>) =_{\{error\} \oplus D} v$

(A2c) $\boldsymbol{hd}(<v>\bullet v^*) =_{\{error\} \oplus D} v$

(A2d) $\boldsymbol{tl}(<v>\bullet v^*) = v^*$

(A2e) $\boldsymbol{hd}(v^*)\bullet\boldsymbol{tl}(v^*) = v^*$

Interpretation

Language Elements	Defined Language	Defining Language
$\mathcal{I}$: domain $\rightarrow$ domain	$\mathcal{I}$(id) = id	
	$\mathcal{I}$(exp) = ecode	
	$\mathcal{I}$(com) = code	
	$\mathcal{I}$(prog) = pcode	
	$\mathcal{I}$(decl) = dcode	
	$\mathcal{I}$(bas) = bas	
	$\mathcal{I}$(opr) = ocode	
		$\mathcal{I}$(num) = num
		$\mathcal{I}$(bool) = bool
		$\mathcal{I}$(loc) = loc
		$\mathcal{I}$(bv) = bv
		$\mathcal{I}$(rv) = rv
		$\mathcal{I}$(dv) = loc $\oplus$ rv $\oplus$ $\mathcal{I}$(proc)
		$\mathcal{I}$(sv) = sv
		$\mathcal{I}$(file) = file
		$\mathcal{I}$(env) = alist $\otimes$ {<>}
		$\mathcal{I}$(store) = store
		$\mathcal{I}$(state) = $\mathcal{I}$(env) $\otimes$ store $\otimes$ ans $\otimes$ stack $\otimes$ dump
		$\mathcal{I}$(ans) = ans
		$\mathcal{I}$(cont) = mcont
		$\mathcal{I}$(dcont) = $\mathcal{I}$(env) $\rightarrow$ mcont
		$\mathcal{I}$(econt) = $\mathcal{I}$(dv) $\rightarrow$ mcont
		$\mathcal{I}$(proc) = mcont $\rightarrow$ $\mathcal{I}$(econt)

I: function symbol → term	*I*(**B**) = [**loadv**, **B**]	*I*(***E***) = λEk. ***H*** (E) (k) where ***H***(C)(λe(r, s, a, stk, d). F) equals ***ME***(C)(λ(r, s, a, stk, d). F(***hd***(stk))((r, s, a, ***tl***(stk), d))
		I(***C***) = λCc. ***MC*** (C) (c)
	I(**true**) = [**loadb**, TRUE]	*I*(***P***) = λP. ***MP*** (P) z_o, where z_o = λ(r,s,a,stk,d). (r,s,a,stk,d)
	I(**false**) = [**loadb**, FALSE]	*I*(***hd***) = λs. ***hd***(s)
	I(**read**) = [**read**]	*I*(***tl***) = λs. ***tl***(s)
	I($\mathbf{I_i}$) = [**load**, $\mathbf{I_i}$], i ≥ 1	*I*(***R***) = λEk. ***H*** (E) (deref^T (rv?^{T} (k)))
	I(__(__)) = λE_1E_2. (E_1) • (E_2) • [**pcall**]	*I*(***D***) = λDu. ***G*** (D) (u) where ***G***(D)(λr′(r, s, a, stk, d). F) equals ***MD***(D)(λ(r, s, a, stk, d). F(pr1(r))((pr2(r), s, a, stk, d)))
	I(**O**) = λE_1E_2. (E_1) • [**deref**] • (E_2) • [**deref**] • [**ocode**]	*I*(***B***) = λB. ***B*** (B)
	I(**:=**) = λE_1E_2. (E_1) • (E_2) • [**deref**] • [**store**]	*I*(***O***) = λo. ***O*** (o)
	I(**output**) = λE. (E) • [**deref**] • [**output**]	
	I(**if**) = λEC_1C_2. (E) • [**deref**] • [**cond**, C_1, C_2]	
	I(**while**) = λEC. [**loop**, E, C]	
	I(**;**) = λC_1C_2. (C_1) • (C_2)	
	I(**begin**) = λDC. [**begin**] • (D) • (C) • [**end**]	
	I(**program**) = λC. [**start**] • (C) • [**halt**]	
	I(**const**) = λIE. (E) • [**deref**] • [**bind** I]	
	I(**var**) = λIE. (E) • [**deref**] • [**init**] • [**bind** I]	
	I(**proc**) = λII_1C. [**mkproc** [**bind** I_1] • C • [**ret**]] • [**bind** I]	
	I(**,**) = λD_1D_2. D_1 • D_2	

predicate symbol → predicate symbol		*I*($=_{econt}$) = $=_{I(econt)}$
		I($=_{ans}$) = $=_{ans}$
		I($=_{id}$) = $=_{id}$
		I(null) = null
		etc.

individual variable symbol → term	*I*(k: econt) = λe(r, s, a, stk, d). (z)((r, s, a, e • stk, d)): (value → mstate → ans) *I*((r,s,a): state) = (r,s,a,stk,d): mstate *I*(c: cont) = z: mcont *I*((<>, s_0, a_0): initial state) = (<>,s_0, a_0, <>, <>): initial mstate *I*(c_0: cont) = z_0: mcont *I*(u: dcont) = λr′(r, s, a, stk, d). (z)((<<r′, pr1(r)>, pr2(r)>, s, a, stk, d)) *I*(p: proc) = λz(r, s, a, stk, d). x(λ(r, s, a, stk, d)). z((***hd***(d), s, a, stk, ***tl***(d)))) ((r, s, a, stk, r•d)) where x: mproc and z: mcont
new predicates	is-econt: (value → mstate → ans) → bool is-cont: mcont → bool etc.

Abbreviations

loc?: econt → econt
loc? = λke. isloc(e) → k(e), (λ(r, s, a). <r, s, <a, error>>)
I(loc?) = λze(r,s,a,stk,d). isloc(e) → z((r,s,a,<e•stk>,d)), <r, s, <a, error>, stk, d>
= loc?^T

proc?: econt → econt
proc? = λke. isproc(e) → k(e), (λ(r, s, a). <r, s, <a, error>>)
I(proc?) = λze(r,s,a,stk,d). isproc(e) → z((r,s,a,<e•stk>,d)), <r, s, <a, error>, stk, d>
= proc?^T

rv?: econt → econt
rv? = λke. isrv(e) → k(e), (λ(r, s, a). <r, s, <a, error>>)
I(rv?) = λze(r,s,a,stk,d). isrv(e) → z((r,s,a,<e•stk>,d)), <r, s, <a, error>, stk, d>
= rv?^T

bool?: econt → econt

```
bool? = λke. isrv(e) → (isbool(e) → k(e), (λ(r, s, a). <r, s, <a, error>>)),
                        (λ(r, s, a). <r, s, <a, error>>)
I(bool?) = λze(r,s,a,stk,d). isrv(e) → (isbool(e) → z((r,s,a,<e•stk>,d)),
                                   <r, s, <a, error>, stk, d>),
                            <r, s, <a, error>, stk, d>
           = bool?^T

update: loc → cont → econt
update = λlce(r, s, a). issv(e) → c(<r, s[e/l], a>), <r, s, <a, error>>
I(update) = λlze(r, s, a, stk, d). issv(e) → z(<r, s[e/l], a, stk, d>),
                                              <r, s, <a, error>, stk, d>
        = update^T

new: store → (loc ⊕ {error})
new = λs. s(l_1) = unused → l_1, ... , s(l_n) = unused → l_n, error
I(new) = new

ref: econt → econt
ref = λke(r, s, a). new(s) = error → <r, s, <a, error>>,
                                        update (new(s)) (k(new(s))) (e) (<r, s, a>)
I(ref) = λze(r, s, a, stk, d). new(s) = error → <r, s, <a, error>, stk, d>,
          update (new(s)) (λ(r, s, a, stk, d). z((r, s, a, new(s) • stk, d)) (<r, s, a, stk, d>)
        = ref^T

deref: econt → econt
deref = λke(r, s, a). isloc(e) → (s(e) = unused → <r, s, <a, error>>, k(s(e))(<r, s, a>)),
                                  k(e)(<r, s, a>)
I(deref) = λze(r, s, a, stk, d). isloc(e) → (s(e) = unused → <r, s, <a, error>, stk, d>,
                            z(<r, s, a, s(e) • stk, d>)), z(<r, s, a, e • stk, d>)
        = deref^T
```

Example Correctness Proof

Axiom (E1)

Translate Axiom into L_{target}

$\boldsymbol{E} \llbracket \mathbf{B} \rrbracket (k) =_{cont} k(\boldsymbol{B} \llbracket \mathbf{B} \rrbracket)$

(translate axiom using interpretation. $\mathcal{I}$)

$\lambda Ek.\ [\boldsymbol{H}(E)\ (k)]\ (\mathcal{I}(\mathbf{B}))\ (\mathcal{I}(k)) =_{mcont} \mathcal{I}(k)(\mathcal{I}(\boldsymbol{B} \llbracket \mathbf{B} \rrbracket))$

(simplify)

$\boldsymbol{H}([\mathbf{loadn},\ \mathbf{B}])\ (\ \lambda e(r,\ s,\ a,\ stk,\ d).\ z((r,\ s,\ a,\ e \bullet stk,\ d))\ ) =_{mcont}$
$(\lambda e(r,\ s,\ a,\ stk,\ d).\ z((r,\ s,\ a,\ e \bullet stk,\ d))\ )(\boldsymbol{B} \llbracket \mathbf{B} \rrbracket)$

(simplify)

$\boldsymbol{ME}([\mathbf{loadn},\ \mathbf{B}])\ (z) =_{mcont} \lambda(r,s,a,stk,d).\ [z((r,s,a,\boldsymbol{B} \llbracket \mathbf{B} \rrbracket \bullet stk,d))]$

Proof in T_{target}

$\boldsymbol{ME}([\mathbf{loadn},\ \mathbf{B}])\ (z)$

(axiom TE2)

$= \boldsymbol{MI} \llbracket [\mathbf{loadn},\ \mathbf{B}] \rrbracket\ (\boldsymbol{ME} \llbracket <> \rrbracket\ (z))$

(axiom TE1)

$= \boldsymbol{MI}([\mathbf{loadn},\ \mathbf{B}])z$

(axiom I1)

$= \lambda(r,s,a,stk,d).\ [z((r,s,a,<\boldsymbol{B} \llbracket \mathbf{B} \rrbracket> \bullet stk,d))]$

Axiom (E2a)

Translation and proof are similar to those for Axiom (E1a).

Axiom (E2b)

Translation and proof are similar to those for Axiom (E1a).

Axiom (E3)

Translate Axiom into L_{target}

E ⟦ **read** ⟧ (k) $=_{econt}$ λ(r,s,a). [null(s(input)) → <r, s, <a, error>>,
k(***hd***(s(input)))((r, s[***tl***(s(input))/input], a))]

(translate axiom using interpretation, ***I***)

λEk. [***H*** (E) (k)] ⟦ ***I***(**read**) ⟧ ***I***(k) $=_{mcont}$ ***I***(λ(r,s,a). [null(s(input)) → <r, s, <a, error>>,
k(***hd***(s(input)))((r, s[***tl***(s(input))/input], a))])

(simplify)

ME ([**read**]) (z) $=_{mcont}$ λ(r, s, a, stk, d). [null(s(input)) → <r, s, <a, error>>, stk, d>,
z((r, s[***tl***(s(input))/input], a, ***hd***(s(input))•stk, d))]

Proof in T_{target}

ME ([**read**]) (z)

(axiom TE2)

= ***MI*** ⟦ **[read]** ⟧ (***ME*** ⟦ <> ⟧ (z))

(axiom TE1)

= ***MI*** ⟦ **[read]** ⟧ z

(axiom I5)

= λ(r,s,a,stk,d). [null(s(input)) → <r,s,<a,error>,stk,d>,
z((r,s[***tl***(s(input))/input],a,<***hd***(s(input))>•stk, d))]

Axiom (E4)

Not shown.

Axiom (E7)

Not shown.

Axiom (C1)

Not shown.

Axiom (C2)

Not shown.

Axiom (C3)

Translate Axiom into L_{target}

$\boldsymbol{C}$ ⟦ $E_1(E_2)$ ⟧ $(c) =_{cont}$ $\boldsymbol{E}$ ⟦ E_1 ⟧ (proc? (λp. $\boldsymbol{E}$ ⟦ E_2 ⟧ (p(c))))

(translate axiom using interpretation, $\boldsymbol{I}$)

$\boldsymbol{MC}$ ⟦ $\boldsymbol{I}(E_1)$ • $\boldsymbol{I}(E_2)$ • **[pcall]** ⟧ $(z) =_{mcont}$
$\boldsymbol{ME}$ ⟦ $\boldsymbol{I}(E_1)$ ⟧ ($proc?^T$ (λp. $\boldsymbol{ME}$ ⟦ $\boldsymbol{I}(E_2)$ ⟧ (p(z))))

(expand abbreviation)

$\boldsymbol{MC}$ ⟦ $\boldsymbol{I}(E_1)$ • $\boldsymbol{I}(E_2)$ • **[pcall]** ⟧ $(z) =_{mcont}$
$\boldsymbol{ME}$ ⟦ $\boldsymbol{I}(E_1)$ ⟧ (λe(r, s, a, stk, d).
iproc(e) →
$\boldsymbol{ME}$ ⟦ $\boldsymbol{I}(E_2)$ ⟧ (e(z)) ((r, s, a, e• stk, d)),
(r, s, <a, error>, stk, d))

Proof in T_{target}

$\boldsymbol{MC}$ ⟦ $\boldsymbol{I}(E_1)$ • $\boldsymbol{I}(E_2)$ • **[pcall]** ⟧ (z)

(axioms TC1, TC2, TC3)

$$= \mathbf{ME} \llbracket \mathbf{I}(E_1) \rrbracket (\mathbf{ME} \llbracket \mathbf{I}(E_2) \rrbracket (\mathbf{MI} \llbracket [\mathbf{pcall}] \rrbracket (z)))$$

$$(\mathbf{ME} \llbracket \mathbf{I}(E_1) \rrbracket (\mathbf{ME} \llbracket \mathbf{I}(E_2) \rrbracket (z)) =$$
$$\lambda(r, s, a, stk, d).\ \mathbf{ME} \llbracket \mathbf{I}(E_2) \rrbracket (z)\ (r, s, a, v_1 \bullet stk, d) =$$
$$\lambda(r, s, a, stk, d).\ z\ (r, s, a, v_2 \bullet v_1 \bullet stk, d))$$

$$= \lambda(r, s, a, stk, d).\ \mathbf{MI} \llbracket [\mathbf{pcall}] \rrbracket (z)\ ((r, s, a, v_2 \bullet v_1 \bullet stk, d))$$

(axiom I22)

$$\begin{aligned}
= \lambda(r, s, a, stk, d).\ & \mathbf{lg}(v_2 \bullet v_1 \bullet stk) < 2 \rightarrow \langle r, s, \langle a, error\rangle, v_2 \bullet v_1 \bullet stk, d\rangle, \\
& isproc(v_1) \rightarrow \\
& \qquad v_1\ (z)\ (\langle r, s, a, \langle v_2 \bullet stk\rangle, d\rangle), \\
& \qquad \langle r, s, \langle a, error\rangle, v_2 \bullet v_1 \bullet stk, d\rangle\ ))
\end{aligned}$$

$$(v_1\ (z)\ (\langle r, s, a, \langle v_2 \bullet stk\rangle, d\rangle) =$$
$$\mathbf{ME}\ (\mathbf{I}(E_2)\ (v_1(z))\ ((r, s, a, stk, d)))$$

(stack has 2 values)

$$\begin{aligned}
= \lambda(r, s, a, stk, d).& \\
& isproc(v_1) \rightarrow \\
& \qquad \mathbf{ME}\ (\mathbf{I}(E_2)\ (v_1(z))\ ((r, s, a, stk, d)), \\
& \qquad \langle r, s, \langle a, error\rangle, v_2 \bullet v_1 \bullet stk, d\rangle\ ))
\end{aligned}$$

$$\begin{aligned}
= \lambda(r, s, a, stk, d).\ & \{\lambda(r', s', a', stk', d'). \\
& isproc(hd(stk')) \rightarrow \\
& \qquad \mathbf{ME}\ (\mathbf{I}(E_2)\ (hd(stk')(z))\ ((r', s', a', stk', d')), \\
& \qquad \langle r', s', \langle a', error\rangle, v_2 \bullet stk', d'\rangle\ )) \\
& \qquad \}\ ((r, s, a, v_1 \bullet stk, d))
\end{aligned}$$

$$\begin{aligned}
= \mathbf{ME} \llbracket \mathbf{I}(E_1) \rrbracket\ & \{\lambda(r', s', a', stk', d'). \\
& isproc(hd(stk')) \rightarrow \\
& \qquad \mathbf{ME}\ (\mathbf{I}(E_2)\ (hd(stk')(z))\ ((r', s', a', stk', d')), \\
& \qquad \langle r', s', \langle a', error\rangle, v_2 \bullet stk', d'\rangle\ )) \\
& \qquad \}
\end{aligned}$$

$$(\langle r', s', \langle a', error\rangle, v_2 \bullet stk', d'\rangle =_{state}$$
$$\langle r', s', \langle a', error\rangle, stk', d'\rangle)$$

$$\begin{aligned}
= \mathbf{ME} \llbracket \mathbf{I}(E_1) \rrbracket\ & \{\lambda(r', s', a', stk', d'). \\
& isproc(hd(stk')) \rightarrow \\
& \qquad \mathbf{ME}\ (\mathbf{I}(E_2)\ (hd(stk')(z))\ ((r', s', a', stk', d')), \\
& \qquad \langle r', s', \langle a', error\rangle, stk', d'\rangle\ )) \\
& \qquad \}
\end{aligned}$$

Axiom (C4)

Not shown.

Axiom (C5)

Not shown.

Axiom (C6)

Not shown.

Axiom (C7)

Not shown.

Axiom (P1)

Translate Axiom into L_{target}

$$\boldsymbol{P}\ [\![\ \textbf{program}\ C\ ]\!]\ (i) = \boldsymbol{C}\ [\![\ C\ ]\!]\ (\lambda(r, s, a).\ \langle r, s, \langle a, stop\rangle\rangle)\ (r_0, s_0[i/input], a_0)$$

(translate axiom using interpretation, $\mathcal{I}$)

$$\lambda P.\ [\boldsymbol{MP}\ (P)\ z_0]\ [\![\ \mathcal{I}(\textbf{program}\ C)\ ]\!]\ (\mathcal{I}(i)) = \lambda Cc.\ [\boldsymbol{MC}\ (C)\ (c)]\ [\![\ \mathcal{I}(C)\ ]\!]\ \mathcal{I}(\lambda(r, s, a).\ \langle r, s, \langle a, stop\rangle\rangle)\ ((\langle\rangle, s_0, a_0, \langle\rangle, \langle\rangle))$$

(translate axiom using interpretation, $\mathcal{I}$)

$$\boldsymbol{MP}\ [\![\ [\textbf{start}] \bullet \mathcal{I}(C) \bullet [\textbf{halt}]\ ]\!]\ \lambda(r, s, a, stk, d).\ (r, s, \langle a, stop\rangle, stk, d)\ ((r, s, a, stk, d)) = \boldsymbol{MC}\ (\mathcal{I}(C))\ (\lambda(r, s, a, stk, d).\ (r, s, \langle a, stop\rangle, stk, d))\ ((\langle\rangle, s_0, a_0, \langle\rangle, \langle\rangle))$$

Proof in T_{target}

$$\boldsymbol{MP}\ [\![\ [\textbf{start}] \bullet \mathcal{I}(C) \bullet [\textbf{halt}]\ ]\!]\ \lambda(r, s, a, stk, d).\ (r, s, \langle a, stop\rangle, stk, d)\ ((r, s, a, stk, d))$$

(axioms TC1, TC2 and TC3)

$$= \boldsymbol{MI}\ [\![\ [\textbf{start}]\ ]\!]\ (\boldsymbol{MP}\ [\![\ \mathcal{I}(C)\ ]\!]\ (\boldsymbol{MI}\ [\![\ [\textbf{halt}]\ ]\!]\ \lambda(r, s, a, stk, d).\ (r, s, \langle a, stop\rangle, stk, d)))\ ((r, s, a, stk, d))$$

(axiom I14)

= (***MP*** ⟦ ***I***(C) ⟧ (***MI*** ⟦ **[halt]** ⟧ λ(r, s, a, stk, d). (r, s, <a, stop>, stk, d))) ((<>,s_o,a_o,<>,<>))

(axiom I15)

= ***MP*** ⟦ ***I***(C) ⟧ λ(r, s, a, stk, d). (r, s, <a, stop>, stk, d) ((<>,s_o,a_o,<>,<>))

(MP-equals-MC lemma)

= ***MC*** ⟦ ***I***(C) ⟧ λ(r, s, a, stk, d). (r, s, <a, stop>, stk, d) ((<>,s_o,a_o,<>,<>))

Axiom (D1)

Translate Axiom into L_{target}

D ⟦ **const** I E ⟧ (u) $=_{cont}$ ***R*** ⟦ E ⟧ (λe. u[e/I])

(translate axiom using interpretation, ***I***)

MD ⟦ ***I***(E) • **[deref]** • **[bind** I] ⟧ (z) $=_{mcont}$
ME ⟦ ***I***(E) ⟧ (deref^T (rv?^{T} (λe(r, s, a, stk, d). z((<<I, e>, r>, s, a, stk, d)))))

Proof in T_{target}

MD ⟦ ***I***(E) • **[deref]** • **[bind** I] ⟧ (z)

(axioms TD1, TD2 and TD3)

= ***ME*** ⟦ ***I***(E) ⟧ (***MI*** ⟦ **[deref]** ⟧ (***MI*** ⟦ **[bind** I] ⟧ (z)))

(axiom I19)

= ***ME*** ⟦ ***I***(E) ⟧ (***MI*** ⟦ **[deref]** ⟧ (
λ(r, s, a, stk, d). ***lg***(stk) < 1 → <r, s, <a, error>, stk, d>,
z(<<<<I, ***hd***(stk)>, pr1(r)>, pr2(r)>, s, a, ***tl***(stk), d>))))

(axiom I16)

= ***ME*** ⟦ ***I***(E) ⟧ (
λ(r, s, a, stk, d). isloc(***hd***(stk)) →
[s(***hd***(stk)) = **unused** → (r, s, <a, error>, stk, d),
[isrv(s(***hd***(stk))) → z′(<r,s,a,<s(***hd***(stk)) • ***tl***(stk)>,d>),
<r, s, <a, error>, stk, d>)],
[isrv(***hd***(stk)) → z′(<r,s,a,stk,d>), <r, s, <a, error>, stk, d>])

where z′ = λ(r, s, a, stk, d). ***lg***(stk) < 1 → <r, s, <a, error>, stk, d>,
z(<<<<I, ***hd***(stk)>, pr1(r)>, pr2(r)>, s, a, ***tl***(stk), d>)

(abbreviations)

= ***ME*** ⟦ ***I***(E) ⟧ (deref^T (rv?^{T} (z′)))

(STACK-HAS-ONE lemma)

ME ⟦ ***I***(E) ⟧ (deref^T (rv?^{T} (λe(r, s, a, stk, d). z((<<I, e>, r>, s, a, stk, d)))))

Axiom (D2)

Translate Axiom into L_{target}

D ⟦ **var** I E ⟧ (u) $=_{cont}$ ***R*** ⟦ E ⟧ (ref (λe. u[e/I]))

(translate axiom using interpretation, ***I***)

MD ⟦ ***I***(E) • **[deref]** • **[init]** • **[bind** I] ⟧ (z) $=_{mcont}$
ME ⟦ ***I***(E) ⟧ (deref^T (rv?^{T} (ref^T (λe(r, s, a, stk, d). z((<<I, e>, r>, s, a, stk, d))))))

Proof in T_{target}

MD ⟦ ***I***(E) • **[deref]** • **[init]** • **[bind** I] ⟧ (z)

(axioms TD2, TD2 and TD3)

= ***ME*** ⟦ ***I***(E) ⟧ (***MI*** ⟦ **[deref]** ⟧ (***MI*** ⟦ **[init]** ⟧ (***MI*** ⟦ **[bind** I] ⟧ (z))))

(axiom I19)

= ***ME*** ⟦ ***I***(E) ⟧ (***MI*** ⟦ **[deref]** ⟧ (***MI*** ⟦ **[init]** ⟧ (
λ(r, s, a, stk, d). ***lg***(stk) < 1 → <r, s, <a, error>, stk, d>,
z(<<<<I, ***hd***(stk)>, pr1(r)>, pr2(r)>, s, a, ***tl***(stk), d>)))))

(axiom I20)

```
= ME ⟦ I(E) ⟧ (MI ⟦ [deref] ⟧ (
λ(r, s, a, stk, d). isloc(new(s)) → z''(<r, s[hd(stk)/new(s)], a, <new(s) • tl(stk)>, d>),
                                    <r, s, <a, error>, stk, d>

where z'' = λ(r, s, a, stk, d).  lg(stk) < 1 → <r, s, <a, error>, stk, d>,
                                 z(<<<<I, hd(stk)>, pr1(r)>, pr2(r)>, s, a, tl(stk), d>)))))
```

(axiom I16)

```
= ME ⟦ I(E) ⟧ (
λ(r, s, a, stk, d).  isloc(hd(stk)) →
            [s(hd(stk)) = unused → (r, s, <a, error>, stk, d),
                                    (isrv(s(hd(stk))) → z'(<r,s,a,<s(hd(stk)) • tl(stk)>,d>),
                                                       <r, s, <a, error>, stk, d>)],
            [isrv(hd(stk)) → z'(<r,s,a,stk,d>), <r, s, <a, error>, stk, d>])

where z' = λ(r, s, a, stk, d).  lg(stk) < 1 → <r, s, <a, error>, stk, d>,
                                z''(<<<<I, hd(stk)>, pr1(r)>, pr2(r)>, s, a, tl(stk), d>)
```

(abbreviations)

```
= ME ⟦ I(E) ⟧ (deref^T (rv?^T (ref^T ((z')))
```

(STACK-HAS-ONE lemma)

```
ME ⟦ I(E) ⟧ (deref^T (rv?^T (ref^T (λe(r, s, a, stk, d).  z((<<I, e>, r>, s, a, stk, d))))))
```

Axiom (D3)

Translate Axiom into L_{target}

```
D ⟦ proc I I_1 C ⟧ (u) =_cont λ(r,s,a).  u[(λce(r',s',a').  C(C)(c)(<r[e/I_1],s',a'>))/I](<r,s,a>)
```

(translate axiom using interpretation, I)

```
MD ⟦ [mkproc [bind I_1] • I(C) • [ret] ] • [bind I] ⟧ (z) =_mcont
λ(r, s, a, stk, d).
(λr'(r, s, a, stk, d). z((<<r', pr1(r)>, pr2(r)>, s, a, stk, d)) )
    <I, λz(r'', s'', a'', stk'', d'').
         (λze(r', s', a', stk', d'). MC ⟦ I(C) ⟧ (z) (<<<I_1, e>, pr1(r)>, pr2(r)>, s', a', stk', d'))
        (λ(r, s, a, stk, d). z(hd(d), s, a, stk, tl(d)) )
        ((r'', s'', a'', stk'', r''•d'')) >
((r, s, a, stk, d))
```

(simplify)

MD ⟦ [**mkproc** [**bind** I_1] • *I*(C) • [**ret**]] • [**bind** I] ⟧ (z) $=_{mcont}$
λ(r, s, a, stk, d).
(λr'(r, s, a, stk, d). z((<<r', pr1(r)>, pr2(r)>, s, a, stk, d)))
<I, λz(r'', s'', a'', stk'', d''). ***MC*** ⟦ *I*(C) ⟧
(λ(r, s, a, stk, d). z(***hd***(d), s, a, stk, ***tl***(d)))
(<<<I_1, ***hd***(stk'')>, pr1(r)>, pr2(r)>, s'', a'', ***tl***(stk''), r''•d'')
((r, s, a, stk, d))

(simplify)

MD ⟦ [**mkproc** [**bind** I_1] • *I*(C) • [**ret**]] • [**bind** I] ⟧ (z) $=_{mcont}$
λ(r, s, a, stk, d).
z((<<<I, λz(r'', s'', a'', stk'', d''). ***MC*** ⟦ *I*(C) ⟧
(λ(r, s, a, stk, d). z(***hd***(d), s, a, stk, ***tl***(d)))
(<<<I_1, ***hd***(stk'')>, pr1(r)>, pr2(r)>, s'', a'', ***tl***(stk''), r''•d'')>, pr1(r)>, pr2(r)>,
s, a, stk, d))

Proof in T_{target}

MD ⟦ [**mkproc** [**bind** I_1] • *I*(C) • [**ret**]] • [**bind** I] ⟧ (z)

(axioms TD3, TD2 and TD3)

= ***MI*** ⟦ [**mkproc** [**bind** I_1] • *I*(C) • [**ret**]] ⟧ (***MI*** ⟦ [**bind** I] ⟧ (z))

(axiom I19)

= ***MI*** ⟦ [**mkproc** [**bind** I_1] • *I*(C) • [**ret**]] ⟧ (z')

where z' = λ(r, s, a, stk, d). ***lg***(stk) < 1 → <r, s, <a, error>, stk, d>,
z((<<<I, ***hd***(stk)>, pr1(r)>, pr2(r)>, s, a, ***tl***(stk), d))

(axiom I21)

= λ(r, s, a, stk, d). z'(<r, s, a,
<λz(r',s',a',stk',d'). ***MP***([**bind** I_1] • *I*(C) • [**ret**])(z)(r,s',a',stk',<r',d'>)> • stk, d>)

(axioms TP1, TP2, TP3, I19, I23)

= λ(r, s, a, stk, d). z'(<r, s, a,
<λz(r',s',a',stk',d'). lg(stk') < 1 → (r',s', <a', error>, stk', d'),
MC (*I*(C)) (λ(r, s, a, stk, d). z((***hd***(d), s, a, stk, ***tl***(d)))
((<<<I_1, ***hd***(stk')>, pr1(r)>, pr2(r)>, s', a', ***tl***(stk'), r'•d')) >
• stk, d>)

(simplify)

= λ(r, s, a, stk, d). z((<<<I,
<λz(r',s',a',stk',d'). lg(stk') < 1 → (r',s', <a', error>, stk', d'),
MC (***I***(C)) (λ(r, s, a, stk, d). z((***hd***(d), s, a, stk, ***tl***(d)))
((<<<I_1, ***hd***(stk')>, pr1(r)>, pr2(r)>, s', a', ***tl***(stk'), r'•d')) >, pr1(r)>, pr2(r)>,
s, a, stk, d))

(lg(stk') ≥ 1 because a procedure is always called with an actual parameter or else an error is returned.)

= λ(r, s, a, stk, d). z((<<<I,
<λz(r',s',a',stk',d').
MC (***I***(C)) (λ(r, s, a, stk, d). z((***hd***(d), s, a, stk, ***tl***(d)))
((<<<I_1, ***hd***(stk')>, pr1(r)>, pr2(r)>, s', a', ***tl***(stk'), r'•d')) >, pr1(r)>, pr2(r)>,
s, a, stk, d))
