FORMALIZING REFERENCE TYPES IN NUPRL

A Dissertation
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Pavel G Naumov
August 1998

FORMALIZING REFERENCE TYPES IN NUPRL

Pavel G Naumov, Ph.D.
Cornell University, 1998

This dissertation defines a Type Theory based semantics for Java-like reference type constructors. The primary focus is made on finding an adequate axiomatization of reference types in Type Theory.

An extension of Type Theory, called *Reference Type Theory*, is introduced. It adds to the Type Theory language a reference type constructor and operations on reference type elements as primitive notions. The dissertation provides informal graph-based semantics for the Reference Type Theory, describes inference rules for this theory, and proves their consistency.

Reference Type Theory is formalized in the Nuprl Proof Development System. This formalization is used to define a formal semantics for a fragment of the Java programming language and to verify several simple Java programs.

BIOGRAPHICAL SKETCH

Pavel Naumov was born on January 26, 1970 in Moscow, Soviet Union. Coming from a family with two generations of researchers in Biology, Chemistry, Physics, Engineering, and Medicine, he was raised to be excited about science. But unlike his younger brother, who followed the route of his parents and became a micro-biologist, Pavel was in love with Mathematics even before he entered elementary school. During his school years he won many mathematical contests including the first prize of the Moscow City Mathematical Olympics two years in a row and a second prize at the Soviet Union Mathematical Olympics. From 1984 till 1987 he attended one of the Moscow schools for students gifted in Mathematics and Physics. There he met his wife-to-be, Elena.

In 1987 he was admitted to the undergraduate program at the Mathematics Department of Moscow State University. In two years, when he and his classmates were required to select a specialization in Mathematics, Pavel's choice of Mathematical Logic was not unexpected – he had read the Russian translation of Mendelson [25] in high school. In 1992, he graduated from Moscow State University with an Honors Diploma and was admitted to the graduate program in Mathematical Logic at the same university. During these years, working under the supervision of Professor Sergei Artemov, Pavel published three articles, presented his work at several Russian and international conferences, and attended several Summer schools in Western Europe.

His fascination with computers started in 1985. One day he went to school leaving running at home one of the first Russian programmable calculators. It took the machine about 8 hours to compute all the digits in the decimal representation of 2^{70} using only 14 registers of available memory. Later there were Pascal and Fortran at high school; PL, Object Oriented Programming concept, λ-calculus, Complexity Theory, and Curry-Howard isomorphism at the university. But it was only in 1993 that Pavel decided to come to Cornell to make Computer Science his profession. During his four years at Cornell, Pavel worked with Professor Robert Constable on formalizing in Nuprl different models of computability. Some of the results are presented in this dissertation.

ACKNOWLEDGMENTS

In spite of my intention to continue life-long self-improvement, this dissertation ends the twenty one years of my formal education. I would like to use this occasion to thank the people who influenced me the most during this time.

I would like to thank my mother, Taisia, who was my most patient listener and my father, Gennadi, who was always a living example of a dedicated scientist. I would like to thank my grandmother, Vera, who helped me to develop a sense of honesty.

I would like to thank my wife, Elena, the only person who can convince me of practically anything in just a few minutes. So, after fourteen years that we know one another I am already not sure if any of my beliefs are original.

I would like to thank my high school Physics teacher, Igor G. Lisenker, who taught me not only Physics, but also how to study and how to teach. I would like to thank my advisor at Moscow, Professor Sergei Artemov, who exposed me to the charm of Provability Logic, and who also managed to arouse my interest in Applied Logic that I had thought never would be possible. I would like to thank Lev D. Beklemishev for discussions that helped my learning Logic.

Of course, special thanks go to my advisor Professor Robert Constable who introduced me to the joy of automatic theorem proving and always encouraged me in my work. Also my thanks to my other two Special Committee members: Professor Tim Teitelbaum, who positively influenced my work by his critical remarks, and Professor Richard Shore for fast but very careful reading of the draft of this dissertation and making many stylistical corrections.

I would like to thank the people at Cornell who helped me to learn Nuprl. Paul Jackson spent many hours with me answering all my questions about the system. Even after Paul left Ithaca, he continued answering my questions by e-mail, while I was learning Nuprl by studying his proofs and tactics. Richard Eaton always was very fast in answering my questions and fixing minor bugs that I happened to find. Karl Crary discussed with me his work on Bar Types and Jason Hickey provided me with the parametrized recursive type rules that he has developed for Nuprl Light.

I would like also to thank the people of the United States for my feeling of being at home from my first step on the American land, as well as for their financial support in the form of ONR N00014-92-J-1764 and DARPA f30602-95-1-0047 grants.

TABLE OF CONTENTS

Biographical Sketch . . . iii
Acknowledgments . . . iv
Table of Contents . . . v
List of Figures . . . vii

1 Introduction **1**
1.1 Background . . . 1
1.2 Goal . . . 1
1.2.1 Java Reference Types . . . 1
1.2.2 Nuprl . . . 2
1.2.3 Methods . . . 2
1.3 Results . . . 3
1.4 Related Works . . . 3
1.4.1 Recursive and Graph Types . . . 3
1.4.2 Java Semantics . . . 4
1.4.3 Web Publishing . . . 4
1.5 Dissertation Outline . . . 4

2 J Programming Language **5**
2.0.1 J Primitive Types . . . 5
2.0.2 Java Standard Class Library . . . 5
2.0.3 J Reference Types . . . 5
2.0.4 Beyond J . . . 6

3 Reference Type Theory **7**
3.1 $RecPair(A)$ Type Constructor . . . 7
3.1.1 Java `RecPairA` Class . . . 7
3.1.2 The Graph Model . . . 8
3.1.3 Naïve Approach . . . 9
3.1.4 Constructor $init(a)$ . . . 9
3.1.5 Destructors $pr_a(p)$ and $pr_r(p)$ . . . 10
3.1.6 Java Assignment Statements . . . 11
3.1.7 Operation $update_a(x, b, y)$ . . . 12
3.1.8 Operation $update_r(x, w, y)$ . . . 14
3.1.9 Canonical Elements of Type $RecPair(A)$ . . . 16
3.1.10 Modeling Java Constructor `new` . . . 17
3.2 Recursive Types . . . 17
3.2.1 Inductive type . . . 18
3.2.2 Co-Inductive Types . . . 18
3.3 Class Type . . . 19
3.3.1 Class Signature . . . 19
3.3.2 The Graph Model . . . 20
3.3.3 Type $\rho(\mathcal{S}@n)$ . . . 20
3.3.4 Destructors $core$ and ref . . . 22
3.3.5 Element Constructor $init$ . . . 22
3.3.6 Operation $update_c(x, b, y)$ . . . 22
3.3.7 Operation $update_r(x, i, z, y)$ . . . 23
3.3.8 Canonical Elements . . . 23
3.3.9 On Modeling `new` constructor . . . 24

4 Formalization in Nuprl 25
4.1 Nuprl Type Theory . . . 25
4.1.1 Basic Types . . . 25
4.1.2 Type Constructors . . . 25
4.2 On Consistency Proofs . . . 28
4.3 $RecPair(A)$ Type in Nuprl . . . 28
4.4 Class Type . . . 30
4.4.1 Class Signature . . . 30
4.4.2 Class Definition . . . 31
4.5 J Class Type . . . 34
4.5.1 J Types . . . 34
4.5.2 J Class Signature . . . 36
4.5.3 J Class Definition . . . 39
4.6 J Semantics . . . 40
4.6.1 Environment . . . 40
4.6.2 Expression . . . 41
4.6.3 Statement . . . 42
4.7 Verification . . . 43

A Deformalization 45
A.0.1 Proof Transformation . . . 45
A.0.2 Proof Publishing . . . 45

B Nuprl Formal Theory Printouts 47
`parec` . . . 48
`signature` . . . 50
`class` . . . 54
`j_type` . . . 75
`j_signature` . . . 80
`j_class` . . . 88
`env` . . . 92
`exp` . . . 93
`stmt` . . . 97
`hoare` . . . 103
`verify_1` . . . 110
indent `rec_pair` . . . 111
Bibliography . . . 114

LIST OF FIGURES

3.1 Graph model for type $RecPair(\mathbb{Z})$ 9
3.2 Element $init(151)$ of type $RecPair(\mathbb{Z})$ 10
3.3 Two different $RecPair(\mathbb{Z})$ elements with equal components 10
3.4 Environment Diagram 12
3.5 Constructor `new` in Type Theory 18
3.6 Graph model for a parametrized class type 21
3.7 Initial Elements 23

A.1 Front page to the Web presentation of a formal theory 46
A.2 Proof step in the Web presentation of a formal theory 46

Chapter 1

Introduction

1.1 Background

From the Leibniz's dream three centuries ago ([23], [22]), *machinis spiritualibus* came to the everyday reality of modern age, helping solve classical mathematical problems ([2], [24], [20]), verify microprocessors ([34]), and formalize a large part of Mathematics ([17], [38]). As machines gain more power and software acquires more intelligence, not only do computers become more useful in verifying existing and producing new knowledge, but they can also assist people in presenting material in a readable, understandable, and appealing way. This can be seen in almost all aspects of our life from personal organizers to online library catalogs.

In Mathematics, the place where knowledge is presented in one of the most compressed and yet elegant ways, computers have also been long and successfully used to make the presentation better. D. Knuth's TeX [19] has changed the way mathematical knowledge is being formulated, presented, and, sometimes, even thought of, by relegating to the computer many typesetting functions. L. Lamport's LaTeX [21] took more control from mathematicians, leaving enforcement of style and handling of the reference structure up to the computer.

The time is coming when machines will control the content of mathematical texts in addition to their form. By controlling the content, not only will computers verify its logical soundness and assist in making deductions, they also will make text more understandable by being able to show the proof in either detailed or sketchy form depending on the reader's preferences. Computers will eliminate ambiguity in formulas by keeping definitions of shortcut notations in the background. They will assist in access to abstractions, theorems, and logical rules used in the text. Computers will adapt the text to the reader's intelligence, background, and native language.

Many of these features are already available in such modern formal proof development environments as Nuprl [9] and CtCoq [37]. Although these systems are not ready yet to replace LaTeX in mainstream Mathematics, they can show how this discipline will look in the next millennium.

1.2 Goal

The goal of this dissertation is to show that formal systems can already be used to produce a readable presentation of advanced, non-obvious, material. I have chosen this material to be semantics of a fragment of a real programming language. The reason is to demonstrate the power of formalization technique on a mathematically rich, but still nearly real world example. In addition, I wanted this dissertation to be the culmination of my three-year work on formal models of computability (deterministic automata [5], non-deterministic automata, Turing machines [29], and Simple Imperative Programming Language [28]).

1.2.1 Java Reference Types

There are several reasons for choosing Java [11] as the programming language for this demonstration. First of all, Java is a well-defined language with a completely specified syntax and relatively simple and clear semantics. Secondly, Java provides an elegant and safe way to deal with references by moving pointers from the level of language syntax to the level of language implementation[1]. On the syntax level Java is using reference type elements to represent references in abstract data structures. These reference types have no direct analogy either in Type Theory – the logical framework that most contemporary theorem provers use, or in the Set Theory – the widely accepted logical foundation of today's informal Mathematics. As a part of my work I wanted to develop a mathematical model of reference types, a model general enough to be used outside of the Java semantics. Thirdly, I hoped that being popular, Java will attract more interest to my work and the power of Formal Mathematics.

1.2.2 Nuprl

The main tool used in the formalization part of this work is the Nuprl Proof Development System [9]. The advantages of using Nuprl as opposed to other formal environments such as HOL [10], PVS [32], and many others are: its rich Type Theory, proof readability, and the executability of its results.

Nuprl Type Theory, briefly described in the beginning of the fourth chapter, provides powerful type constructors such as parametrized recursive types and bar types that were essential for my work. In fact, as a part of my work on Java semantics, I have added parametrized recursive types to the Nuprl system based on P. Mendler's work [26].

Unlike most other automated proof systems, Nuprl considers formal proof, not only the statement of the theorem that it supports, to be interesting for people. To help humans to read formal proofs, Nuprl uses a tree-like representation of the proof in computer memory and a feature-rich proof editor for browsing the proof tree. The proof editor is capable of displaying intermediate subgoals, generated after each tactic application. In my work on Web publishing of Formal Mathematics, described in appendix A, I have developed an ML program that converts a Nuprl proof tree into a set of hyper-linked HTML pages, making formal results accessible to practically anyone in the world.

There are two meanings of the statement that Nuprl results are executable. First, any term such as, say, an abstraction, can be executed using the Nuprl built-in term evaluator. Hence, any λ-term, representing a Java expression in my semantics, can be evaluated. This Nuprl feature converts Java semantics into a Java abstract machine capable of program execution. Secondly, since all Nuprl proofs are constructive, a program can be extracted from any existence proof. This program also can be executed using the Nuprl evaluator. Of course, the Nuprl evaluator can not be considered as an adequate substitute for a Java Virtual Machine because the evaluator is not efficient enough.

My intention to define executable semantics determined the choice between operational and denotational semantics. Operational semantics describes a *relation* between the states of an abstract machine, making it harder to evaluate a program. Denotational semantics describes a program as a function that maps the environment before the execution into the environment after the execution. Hence, the program execution can be simulated by a corresponding function evaluation.

Therefore, the simplest way to make semantics executable is to define it as denotational. Unfortunately, the functions that represent Java programs in denotational semantics are not necessarily total, but all functions in Nuprl Type Theory are total. As a result, I needed to use Nuprl bar types ([33], [6]) to represent partial functions. Pros and cons of this approach will be discussed later in the dissertation.

1.2.3 Methods

There are two means by which I wanted to achieve the goal of creating a readable formalization: level structuring and deformalization.

Level structuring is a very powerful tool in creating a well-organized formal, as well as informal, presentation. In formal Mathematics it can be applied in different situations to achieve the same goal – greater simplicity. In abstraction definitions, several similar definitions can have a similar form or can even be based

[1]Being asked why he had chosen to leave pointers in the language syntax creating Pascal, Niklaus Wirth ([1], p. 119) answered that he was not able to find a "flexible" and "effective" way to pursue the pointer-less approach in the time limits he had. Java authors, in my opinion, have found such a way.

on one scheme such as definition by induction or by case split. In order to make the text more understandable, such schemes should be separated into special abstractions. In my formalization I have used this approach by introducing many case split schemes and, for example, by using λ-term for "simple Java binary expression" to define terms for addition, subtraction, multiplication, and boolean operators. On the theorem proving level, structuring means dividing proofs into simpler and more general lemmata, that can potentially be re-used in other theorems. Nuprl well-formness lemmata are the most obvious example of this technique, but the reader will find several other simple lemmata, used extensively throughout the formalization of Java semantics.

In theory design, level structuring means dividing material into smaller theories each of which deals with a separate topic using the most suitable language and the highest possible level of generality. In my work, I introduced the Reference Type Theory as an intermediate level between Java semantics and the standard Type Theory. Reference Type Theory extracts from the Java notion of reference type a simpler and more general concept, that can be added as a new primitive constructor to the Type Theory, or can be defined internally using parametrized inductive types. After studying this general concept of reference types, I have defined a special case of reference types, that corresponds to Java reference types, and have used them to construct a Java semantics.

By deformalization I mean converting a formal theory from an internal computer representation into a form more suitable for reading by a human being. In chapter 4 I deformalize a Java semantics by providing extensive verbal comments for almost each object in the formal library and in appendix A I describe the converter from Nuprl to HTML that I have developed. This converter automatically publishes formal mathematical theories on the Web as sets of hyper-linked HTML pages.

1.3 Results

The main results of this work are a Reference Type Theory and its application to the formal semantics of a Java fragment. As a part of this work I also developed a converter for publishing Formal Theories on the Web.

Reference Type Theory gives a simple and general axiomatization of references in Type Theory. It can be used to provide semantics of other languages or to reason about recursive data structures in general. In particular, it can be the theoretical foundation of a new Nuprl evaluator, more efficient with respect to abstract data types.

The formal executable semantics, based on the Reference Type Theory, clarifies Java reference type structure and may be used for verification of simple Java programs.

The converter is a universal tool for publishing Nuprl formal Mathematics on the Web. It makes formal results widely accessible. This converter and created with its help presentation of formal mathematics on the Web is used by many people at Nuprl group for Web publishing of their own results, teaching, and research.

1.4 Related Works

1.4.1 Recursive and Graph Types

Reference types can be classified as a new form of recursive types. Two other kinds of recursive types, inductive and co-inductive types, have been extensively studied in the past. The literature and the difference between reference, inductive, and co-inductive types will be discussed in chapter 3 after the presentation of the Reference Type Theory.

Reference types were originally introduced in [18] by N. Klarlund and M. Schwartzbach under the name *graph types* using different notations. In this dissertation I define operations *update* and *new*, extend theory with parametrized types, give axiomatization, and prove consistency of Reference Type Theory as an extension of standard Type Theory.

1.4.2 Java Semantics

Earlier works on formalizing Java semantics focused primarily on proving Java type-safety or providing semantics for Java bytecode. D. Oheimb and T. Nipkow ([31], [30]) defined formal operational semantics for a fragment of Java language, which they call Bali, in the Isabelle prover. Their semantics is based on formalization of a Java interpreter in the Type Theory. Each Java object is associated, with some location in the global memory and the reference components of the object are treated as pointers. Similar work was done by D. Syme [36] for another fragment of Java and in [13] for Scheme. Unlike their work, I wanted to represent Java reference types by *types* in Type Theory and to consider objects as elements of these types instead of modeling them by pointers on a Random Access Memory, also I wanted to create a denotational semantics so it would be executable using Nuprl internal evaluator.

R. Cohen [4] has formalized the semantics of a part of the Java Virtual Machine, not Java language, essentially by writing an interpreter in Common Lisp. He used ACL2, the latest version of the Boyer-Moore theorem prover [3]. The other difference between my work and the work mentioned above is that I aimed at producing a *readable* formalization.

1.4.3 Web Publishing

R. Stärk ([35]) converted formal theories of the Logic Program Theorem Prover to HTML. He represents theory content in a similar fashion to my theory overview page, but the only provided hyper-links are pointing to non-structured theorem proofs. The HTML interface to Isabelle formal library [16] includes applets that represent theory dependency graphs and HTML pages with theory ML sources. L. Mikušiak and M. Adámy ([27]) have developed Netscape plug-in for displaying formal mathematics in Z notations. It can display special symbols in different colors and provides, on-demand, auxiliary information about them. Unfortunately, this plug-in available only for $Windows\ 95^{(\mathrm{TM})}$ and $Windows\ NT^{(\mathrm{TM})}$ and is not free for users ([40]).

The first time Nuprl formal theories became available on the Web as a result of the Web interface to the system designed by R. Vaughn and D. Svitavsky ([39]). That presentation was based on interaction with a copy of Nuprl running on the server. It provided access to theorem and abstraction definitions, including map-based access to term structure. Unfortunately, the interface was not very stable because of the internal bugs. It also worked slowly since it relied on a running copy of Nuprl and was using bulky image files to represent formulas. P. Jackson has used similar technique to publish a “static” library of his theories without interaction with any server script. He provided more stable access to the library, but his presentation lacked term structure information and proofs. In addition, Web pages were taking too much time to download. My Nuprl to Web converter, which became available in 1996, was the first to provide static access to formal proofs and to use special symbol images instead of converting the entire formula to a graphic file. It was used by several members of Nuprl group to publish their own theories on the Web. S. Allen has modified this converter to supplement each proof page with the lemma statements and definitions of abstractions used on it.

1.5 Dissertation Outline

The second chapter describes the fragment of the Java language that will be formalized. The third chapter presents the concept and the general theory of reference types. The fourth chapter gives brief introduction to Nuprl and describes the formalization of reference types via parametrized inductive types and formalization of the Java semantics using reference types. The design of the Web converter described in the Appendix A. Formal theory printouts are located in the Appendix B. They show the original form of the material on which chapter 4 is based.

Chapter 2

J Programming Language

This dissertation illustrates concept of Reference Type Theory semantics by defining a formal semantics for a fragment of the Java programming language. The first chapter is dedicated to the description of this fragment. It will be called the *J* programming language.

The semantics will be defined in the next two chapters. This definition will consist of specification of types in Type Theory that correspond to J types, types for λ-terms, representing in Type Theory J expressions, and such terms for sample J expressions. I do not provide a λ-term for every J expression because this becomes a routine procedure after all appropriate types are specified. By the J language I mean a subset of Java syntax for which semantics can be defined in the framework of my type definitions.

2.0.1 J Primitive Types

J includes two Java primitive types: `boolean` and `integer`. J has the standard set of operations on booleans, including *conditional or* and *conditional and*. J `integer` type, as it is defined below, differs from Java integer type because it has infinitely many elements. Existing J integer type can easily be adjusted to handle limited segment of integers by checking the result of every arithmetic operation and throwing appropriate overflow exception each time it is needed. All Java arithmetic operations that return an integer as well as comparison of two integers with the result that has type `boolean` belong to J language.

2.0.2 Java Standard Class Library

The only Java standard class that is included in J is class `Exception`. Equality is the only operation defined on elements of this type. For the purposes of the semantics simplicity, it will be convenient to classify `Exception` as one of the J primitive types.

2.0.3 J Reference Types

J has two reference types: array and class. The array type in J is identical to the Java version except that its length is a J (unbounded) integer.

Type `class` in J is the same as Java class without inheritance. Each Java class is an extension of basic `Object` class. Unless it is defined to be `final`, any Java class can be extended by adding new variables and methods to it. J lacks `Object` class and the extension operator. All J classes are independent and each object belongs to only one J class.

The Java subclass relation cannot be directly modeled by Nuprl subtyping although subclass relation could be potentially added to Reference Type Theory as new primitive predicate. I have not pursued this goal because such an extension will overload the theory and its semantics, making reference type idea less appealing to a potential reader.

J has the same field access operator `e.x` on objects as Java. Any Java method is legal in J as long as it uses only J types and expressions.

2.0.4 Beyond J

Any element of Java syntax that has not been mentioned above does not belong to J. In particular, J does not have

1. `Threads`. They are part of the Java Standard Class Library.
2. random number generator. It is also a part of the Java Standard Class Library.
3. real numbers
4. `char` type. This one easily can be added if needed.

Chapter 3

Reference Type Theory

This chapter introduces a new kind of type into Type Theory – reference type. In the first section I study an example of the reference types – type *RecPair*. In the second section reference types are compared to two other recursive type constructors: inductive and co-inductive types. The last section presents the general theory of reference types.

3.1 $RecPair(A)$ Type Constructor

3.1.1 Java `RecPairA` Class

For any two elements a and b of types A and B respectively, pair $\langle a, b\rangle$ is an element of type $A \times B$, known as the Cartesian product of types A and B. The Cartesian product type comes with the constructor *pair* which for any two elements a and b produces the pair $\langle a, b\rangle$ and two destructors, the *first projector* pr_1 and the *second projector* pr_2, such that

$$pr_1\langle a, b\rangle = a \qquad pr_2\langle a, b\rangle = b. \tag{3.1}$$

The most direct analogy in Java to the Cartesian product of types A and B is Java class

$$\texttt{PairAB class }\{\texttt{A a; B b;}\} \tag{3.2}$$

that for any two Java types `A` and `B` defines a new class `PairAB`[1]. Java constructor

$$\texttt{PairAB(A x, B y)}\{\texttt{a = x; b = y;}\} \tag{3.3}$$

is an analog of the pair constructor in Type Theory and Java expression `p.a` and `p.b` represent the first and the second projectors. In fact, even equations 3.1 hold in Java in the sense that, for instance, expression `PairAB(x,y).a` always returns the same value as expression `x`. But in spite of this similarity, as we will see below, slight modification of the definition 3.2 creates a Java class unknown in the Type Theory world.

Indeed, because class components are implemented in the Java Virtual Machine as pointers, Java allows the name of the class to appear as the type of one or more of its components. For example, for any type `A`[2] we can define class

$$\texttt{RecPairA class }\{\texttt{A a; RecPairA r;}\} \tag{3.4}$$

At first glance, class `RecPairA` also looks similar to the Cartesian product: it has constructor

$$\texttt{RecPairA(A x; RecPairA y)}\{\texttt{a = x; r = y;}\} \tag{3.5}$$

[1] Because of limitations in the Java language syntax we are not able to define parametrized class `Pair(A,B)` uniformly. Instead, we have to define class `PairAB` separately for each pair of Java types `A` and `B`.

[2] We will assume here that `A` is one of primitive Java types such as `int`, `float`, `boolean`, or `char`. More general case, when `A` can be a Java reference type, will be considered in the end of this chapter.

and destructors `p.a` and `p.r` that satisfy equations 3.1. But this similarity ends as soon as we realize that there are objects in class `RecPairA` that have themselves as their second components. For example, for any object `a` of type `A` Java constructor

$$\texttt{RecPairA(A x)}\{\texttt{a = x; r = this;}\} \tag{3.6}$$

returns such an object.

This example illustrates that class `RecPairA` is indeed different from the Cartesian product type because it easily can be shown that in either Type Theory or in Set Theory can a pair can never have itself as one of its components. The same example shows the second major difference between class `RecPair` and Cartesian product – any element of Cartesian product type can be constructed using the *pair* constructor, but not every element of `RecPairA` type can be built by constructor 3.4. In fact, no objects of class `RecPairA` can be created using only constructor 3.4 because it requires an object `y` of the class `RecPairA` before the constructor is applied for the first time.

There are two ways to present class `RecPairA` in Type Theory. The *low-level*, or *indirect*, approach would be to imitate the pointer structure used by the Java Virtual Machine inside Type Theory. Using this approach we would define class `RecPairA` as a function:

$$F : \mathbb{N} \to A \times \mathbb{N}$$

and objects of the class `RecPairA` would be represented by natural numbers. Such an approach assumes that function F is given *a priori* as a part of the type definition. It makes it impossible to construct any "new" elements of this type. Hence, we need to have a very sophisticated function F that includes all possible "objects" that we may need in the future, or we must use the so called *object state* model where function F has an extra argument known as *time* or *machine state*. In this case we can not think about an object as just an element of a type, but need to think about functions on time or machine states. Either way we end up with a much more complicated structure than Cartesian product type that corresponds to class `PairAB`.

The second, *high-level*, or *direct*, approach to presenting class `RecPairA` in Type Theory is to add a new primitive type constructor to Type Theory that satisfies all the properties of class `RecPairA` that have been mentioned above. The major advantage of this approach is its simplicity. It will just map Java classes into appropriate types in Type Theory without any extra functions, time, or states. The second advantage of this approach is that it develops a new constructor for Type Theory that reflects the important concept of reference in typed programming languages.

Of course, there is a price to pay for choosing the second approach. Once a new primitive abstraction is added to the Type Theory we will also need to add new axioms or *inference rules*. The biggest challenge will be to prove that new inference rules are consistent with existing Type Theory.

In this dissertation I develop the second approach in a much more general case than just Java class `RecPairA`, but before doing so I illustrate the main ideas on the sample case of `RecPairA` class. To do so I will add to Type Theory new primitive type constructor $RecPair(A)$[3] that for any type A produces type $RecPair(A)$ that will model Java class `RecPairA` [4].

3.1.2 The Graph Model

Before proceding with the formal description of operations on elements of type $RecPair(A)$ and inference rules that axiomatize these operations, I want to explain the intuitive model behind these axioms. Type $RecPair(A)$ is modeled by an infinite directed graph where there is exactly one edge starting at each vertex. Each vertex is labeled by an element of type A. Different vertices may be labeled by the same element. In this model, every vertex v represents an element of type $RecPair$, the label on vertex v is its a-component and the unique edge starting at vertex v points to element's r-component. Figure 3.1 illustrates a fragment of the graph for type $RecPair(\mathbb{Z})$. It is important to note that there is only one graph representing type

[3] In this chapter I will use `typewriter` style when referring to Java constructors and expressions and *italic* for their Type Theory counterparts.

[4] As has been mentioned above, Java does not allow the definition of parametrized class `RecPair(A)` so we need to define a separate `RecPairA` class for each type `A`. On the other hand, it is natural for Type Theory to have one parametrized type $RecPair(A)$ instead of type constant $RecPairA$ for each type A.

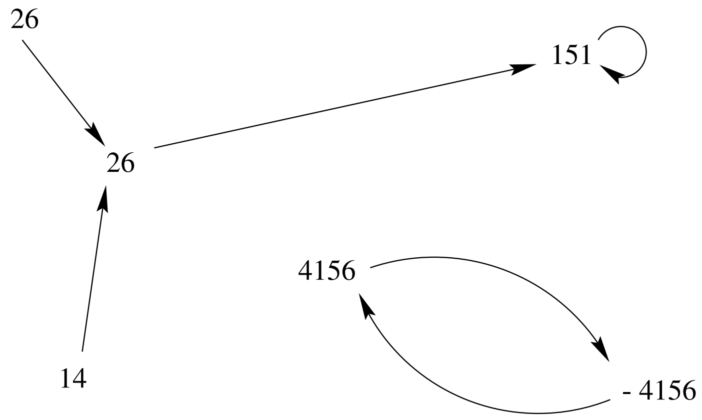

Figure 3.1: Graph model for type $RecPair(\mathbb{Z})$

$RecPair(A)$ for each type A and this graph has infinitely many vertices and edges. Hence, Figure 3.1 displays only a finite fragment of the graph.

Obviously, there are many graphs that satisfy the description above. Not all of them can be used to model $RecPair(A)$ type. Later I will formulate axioms for elements of type $RecPair(A)$ and discuss what kind of restrictions on the graph they impose.

3.1.3 Naïve Approach

In order to add a new type constructor, we not only need to name it, but we also need to specify elements of this type and their relation to elements of other types by selecting the set of primitive operations on the elements and formulating axioms for these operations. The naïve way to do so would be to try to apply the scheme already used for the Cartesian product type: one type constructor and two type destructors that return the components. Let us call such constructor *recpair* and destructors pr_a and pr_r. We want constructor *recpair* to return an element $recpair(a, r)$ of type $RecPair(A)$ for each element a of type A and element r of type $RecPair(A)$. For any element p of type $RecPair(A)$, projectors $pr_a(p)$ and $pr_r(p)$ should return elements of types A and $RecPair(A)$ correspondingly. In addition, we would like to assume that the following version of Cartesian product properties 3.1 hold for type $RecPair(A)$:

$$pr_a(recpair(a, r)) = a \qquad pr_r(recpair(a, r)) = r \tag{3.7}$$

Unfortunately, from our discussion on page 8 it follows that this naïve approach fails because constructor *recpair* can not build all elements of type $RecPair(A)$ and since it needs to have at least one of them in order to construct any new one.

Below we will choose another approach. Instead of the universal constructor $recpair(a, r)$ that was expected to be able to produce any element of type $RecPair(A)$, we will introduce a weaker constructor $init(a)$ that will be able to construct only some elements of type $RecPair(A)$. All other elements of this type will be the results of operations on type $RecPair(A)$ that I will define later.

3.1.4 Constructor $init(a)$

We will assume that for any element a_0 of type A there is at least one element of type $RecPair(A)$ that has a_0 as its a-component and itself as its r-component. We call this self-referential element $init(a_0)$. In our graph model $init(a_0)$ corresponds to a vertex labeled by element a_0 which is an initial and final point of a loop edge. This situation is illustrated on Figure 3.2. If a graph has several self-referential vertices labeled by a_0 any of them can be chosen to be $init(a_0)$. We add an inference rule to the Type Theory [5] that

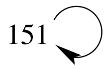

Figure 3.2: Element $init(151)$ of type $RecPair(\mathbb{Z})$

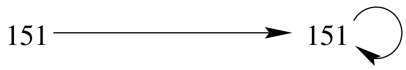

Figure 3.3: Two different $RecPair(\mathbb{Z})$ elements with equal components

postulates the existence of the element $init(a)$ for each element a of type A:

Inference Rule I

$$\frac{a \in A}{init(A) \in RecPair(A)}$$

As one can see, constructor $init$ is just a Type Theory version of the Java constructor 3.6. We will be able to formulate the other properties of the constructor $init$ only after extending the Type Theory language with two destructors of type $RecPair(A)$ elements.

3.1.5 Destructors $pr_a(p)$ and $pr_r(p)$

Like Cartesian product projectors pr_1 and pr_2, $RecPair(A)$ type projectors pr_a and pr_r return corresponding components of the type element. These components have types A and $RecPair(A)$:

Inference Rule II

$$\frac{p \in RecPair(A)}{pr_a(p) \in A}$$

Inference Rule III

$$\frac{p \in RecPair(A)}{pr_r(p) \in RecPair(A)}$$

Because instead of the *pair* constructor we have the *init* constructor, equations 3.1 will have slightly different form in our case:

Inference Rule IV

$$\frac{a_0 \in A}{pr_a(init(a_0)) = a_0}$$

Inference Rule V

$$\frac{a_0 \in A}{pr_r(init(a_0)) = init(a_0)}$$

It is important to remember that unlike elements of the Cartesian product type, elements of $RecPair(A)$ type are not uniquely determined by their components. Figure 3.3 presents two different vertices labeled by the same number and their edges point to the same elements. Obviously, such elements of the type $RecPair(A)$ must be different because one of them is self-referential and the other is not. Although we will

[5] Although in the next chapter I will formalize reference types in Nuprl Type Theory, in this chapter I prefer to formulate "generic" rules that can be adopted to a wide range of Type Theory formalizations.

consider distinct vertices in our graph model to represent different elements of type $RecPair(A)$, discussion of equality on type $RecPair(A)$ definitely should be postponed – at this moment we do not even have sufficient inference rules to prove that there is an element of type $RecPair(\mathbb{Z})$ that can be represented by the left vertex of Figure 3.3.

We will be able to prove this after the introduction of the operations on type $RecPair(A)$. These operations are Type Theory equivalents of assignment statements in Java.

3.1.6 Java Assignment Statements

We intend to find operations on type $RecPair(A)$ that correspond to assignments of a new object component in Java. But before doing so let us consider the effect that is created by a simple assignment of a new value to an integer variable. For example, Java code

$$\texttt{n} = \texttt{5;} \tag{3.8}$$

assigns integer value `5` to variable `n`. In Type Theory terms, this statements assigns an element 5 to variable `n`. A function that maps variable names to elements of appropriate types is called an *environment*. Therefore, we may say that assignment 3.8 changes the environment from env to

$$env' = \lambda v. \begin{cases} 5 & \text{if } v = \texttt{n} \\ env(v) & \text{otherwise} \end{cases}$$

Now let us assume that Java variable `p` has type `RecPairA` and `a0` is a Java constant or variable[6] of type `A`. How does assignment:

$$\texttt{p.a} \ = \ \texttt{a0;} \tag{3.9}$$

change the environment env? Obviously, this assignment changes the value of the variable `p` because its a-component is changed. In the Type Theory terms it means that the statement 3.9 changes the element of $RecPair(A)$ assigned to variable `p` by the environment.

It may be less obvious that the assignment 3.9 also produces other changes in the environment. To show this, assume that the value of variable `p` is Java object x and that there is another variable `q` the va value y of which has x as its second component:

$$env(\texttt{p}) = x$$

$$env(\texttt{q}) = y$$

$$pr_r(y) = x$$

The environment created after evaluation of assignment 3.9, let us call this environment env', maps the variable `p` into a Java object x'. We already know that object x' is different from object x because these two objects have different a-components. We also expect that assignment 3.9 should not change the fact that the value of variable `p` is the r-component of the value of variable `q`:

$$pr_r(env'(\texttt{q})) = env'(\texttt{p}) \tag{3.10}$$

Hence, we can conclude that environments env and env' should assign different values to variable `q` because

$$pr_r(env'(\texttt{q})) = env'(\texttt{p}) = x' \neq x = pr_r(y) = pr_r(env(\texttt{q}))$$

Therefore, assignment 3.9 will change the environment function not only on variable `p`, but also on some other variables of type `RecPairA`.

As it will turn out, function env' can be decomposed into a function

$$h : RecPair(A) \to RecPair(A)$$

and function env:

$$env' = h \circ env \tag{3.11}$$

$$\begin{tikzcd}
\text{Variables} \arrow[r, "env"] \arrow[d, "id"] & RecPair(A) \arrow[d, "h"] \\
\text{Variables} \arrow[r, "env'"] & RecPair(A)
\end{tikzcd}$$

Figure 3.4: Environment Diagram

In other words, we will be able to prove that there is a function h that makes diagram 3.4 commutative.

In order to establish the existence of such a function h we only need to prove that for any two variables `p` and `q`, if $env(\texttt{p}) = env(\texttt{q})$ then $env'(\texttt{p}) = env'(\texttt{q})$. But we know that statement 3.9 does not re-assign any variables. Hence, if two variables pointed to the same object before the evaluation of the statement, then they will point to the same object after the evaluation.

Therefore, we have proved that there is a function h from the type $RecPair(A)$ into itself that satisfies equation 3.11. We will call function h the *update* function because it basically "updates" environment *env* to environment env'.

A similar result can be proven for assignment

$$\texttt{p.r} = \texttt{r0};$$

where `r0` is a Java constant or variable of type `RecPairA`.

3.1.7 Operation $update_a(x, b, y)$

In the previous section we introduced function *update* to represent assignment 3.9 in Type Theory. This assignment has three parameters: variable `p`, class component name `a`, and constant expression `a0`. Note that if `p` and `q` are two different variables that have the same value, then the two assignments

```
p.a = a0;                                   q.a = a0;
```

will have exactly the same effect on the environment. This means that *update* is a function of the variable `p`'s value, not its name. For the same reason, the second argument of *update* function is the value of expression `a0`, not the expression itself.

Also, because type $RecPair(A)$ has only two components, instead of using component name as a function *update* parameter, I prefer to deal with two different update functions $update_a$ and $update_r$. We will start with the function $update_a$, leaving $update_r$ for the next section.

If element x of type $RecPair(A)$ represents in Type Theory the value of variable `p` and element b of type A represents the value of expression `a0`, then $update_a(x, b)$ is a function

$$update_a(x, b) : RecPair(A) \to RecPair(A)$$

It is an empirical observation that we seldom use functions without applying them to some argument[7]. This is why, as with most other Type Theory operators, it is more convenient to consider the three-place operator $update_a(x, b, y)$ that combines two-place operator $update_a(x, b)$ and function application:

$$update_a(x, b, y) := update(x, b)\ y$$

[6] If `a0` is a more complicated expression then its evaluation, if considered as a part of assignment 3.9 evaluation, may arbitrarily change the environment as a side effect.

[7] Ironically, this section will be an exception.

Now we are ready to add *update* as a new primitive to Type Theory. For any two elements x and y of type $RecPair(A)$ and for any element b of type A, expression $update_a(x, b, y)$ has type $RecPair(A)$. This can be formalized as the following inference rule:

Inference Rule VI

$$\frac{x, y \in RecPair(A) \qquad b \in A}{update_a(x, b, y) \in RecPair(A)}$$

After a new primitive abstraction is introduced, we need to add axioms that relate it to other abstractions in the theory. In our case, we want to specify components of element $update_a(x, b, y)$. Because we want type $RecPair(A)$ to model Java class `RecPairA` we again turn to informal arguments about Java in order to find the axioms.

If *env* is the environment before assignment 3.9 is evaluated then

$$env' = \lambda v.update_a(env(x, b, env(v)) \tag{3.12}$$

is the environment after this assignment is evaluated. We know that after the evaluation the a-component of the object, corresponding to variable `p`, is equal to b:

$$b = pr_a(env'(\mathtt{p})) = pr_a(\lambda v.update(x, b, env(v))\ \mathtt{p}) = pr_a(update(x, b, x))$$

This can be formalized as an inference rule

Inference Rule VII

$$\frac{x \in RecPair(A) \qquad b \in A}{pr_a(update_a(x, b, x)) = b}$$

Although we have established this rule only for the case when there is a variable that has value x and there is a variable or constant that has value b, I formulated this rule for any x of type $RecPair(A)$ and any b of type A, because of the fundamental assumption that a Java object not named by a variable has exactly the same properties as objects that are denoted by variables. I will use the same assumption for deriving the other inference rules for type $RecPair(A)$[8].

Similarly, if variable `q` has value y before the evaluation and objects x and y are different, then we know that a-component of variable `q` value will not be changed by assignment 3.9:

$$pr_a(y) = pr_a(env(\mathtt{q})) = pr_a(env'(\mathtt{q})) =$$

$$= pr_a(\lambda v.update_a(env(x), b, env(v)))\ \mathtt{q}) = pr_a(update_a(x, b, y))$$

This justifies the following inference rule:

Inference Rule VIII

$$\frac{x, y \in RecPair(A) \qquad x \neq y \qquad b \in A}{pr_a(update_a(x, b, y)) = pr_a(y)}$$

Situation with the r-component is more complicated since r-component of $update(x, b, y)$ is an element of type $RecPair(A)$ and, as we know, assignment 3.9 can change value of `RecPairA` variable even if this value is different from the one of variable `p`.

Fortunately, we can still say that if before the evaluation variables `p`, `q`, and `t` had values x, y, and z and object z was r-component of object y, then after the evaluation the new value of variable `t` is still the r-component of variable `q` value:

$$pr_r(env'(\mathtt{q})) = env'(\mathtt{t})$$

Using 3.12 the last equation can be reduced to

$$pr_r(update_a(x, b, y)) = update_a(x, b, z)$$

Taking into account that $z = pr_r(y)$, the following inference rule can be stated:

[8] As with any other "fundamental principle", this assumption is true only in some simplified abstract model of the real world. In the Java Virtual Machine objects that are not named directly by variables or indirectly via expressions may be subject to destruction by the garbage collector.

Inference Rule IX

$$\frac{x, y \in RecPair(A) \qquad b \in A}{pr_r(update_a(x, b, y)) = update_a(x, b, pr_r(y))}$$

We also know that because assignment 3.9 does not re-assign variables, for any two Java variables `p` and `q`, $env'(\texttt{p}) = env'(\texttt{q})$ implies that $env(\texttt{p}) = env(\texttt{q})$. This means that function $\lambda y.update_a(x, b, y)$ is a 1-1 mapping:

Inference Rule X

$$\frac{update_a(x, b, y_1) = update_a(x, b, y_2)}{y_1 = y_2}$$

Finally, let us determine what kind of rstriction on our graph model we should impose to garantee that inference rules VII, VIII, IX, and X are satisfied. As one can see, these rules state that function

$$\lambda y.update(x, b, y)$$

is a 1-1 endomorphism of the type $RecPair(A)$ graph that preserves all edges and labels at all vertices except for vertex x. The image of vertex x is labeled by b. In order for a graph to be a model of the type $RecPair(A)$ there should exist a mapping from the graph into itself that satisfies the conditions above.

3.1.8 Operation $update_r(x, w, y)$

In this section I introduce one more operator on the type $RecPair(A)$. This operator corresponds to Java environment update, generated by Java assignment

$$\texttt{p.r = r0;} \tag{3.13}$$

Similarly to the case of assignment 3.9, changes in the environment generated by this assignment can be represented by a function $update_r(x, w, y)$:

$$env' = \lambda v.update_r(x, w, env(v)) \tag{3.14}$$

where x is the value of variable `p` and w is the value of variable `r0`. Formal inference rule that specifies the type of $update_r(x, w, y)$ operator is

Inference Rule XI

$$\frac{x, w, y \in RecPair(A)}{update_r(x, w, y) \in RecPair(A)}$$

Just as in the case of $update_a(x, b, y)$ operator, we will formulate the inference rules for the operator based on our informal intuition about corresponding Java assignment.

We know that for any variable `q` of type `RecPairA` , the a-component of the value of the variable `q` should be the same before and after the evaluation:

$$pr_a(env(\texttt{q})) = pr_a(env'(\texttt{q}))$$

Using 3.14 we can reduce this equation to

$$pr_a(env(\texttt{q})) = pr_a(update_r(x, w, env(\texttt{q})))$$

If y is the value of variable `q` before the evaluation, then

$$pr_a(y) = pr_a(update_r(x, w, env(y)))$$

We have established this fact only for objects x and y that were the values of some variables of type `RecPairA` before the evaluation took place, but in Type Theory we assume this property for any objects x and y of type $RecPair(A)$:

Inference Rule XII

$$\frac{x, w, y \in RecPair(A)}{pr_a(update_r(x, w, y)) = pr_a(y)}$$

We also know that r-component of the value of the variable `p` after the evaluation should be equal to the value of variable `r0` after the evaluation:

$$pr_r(env'(\mathtt{p})) = env'(\mathtt{r0})$$

With the help of 3.14 this equation can be reduced to:

$$pr_r(update_r(x, w, x)) = update_r(x, w, w)$$

Again, we have established the last equality only for objects that are denoted by some variables, but in Type Theory we will assume this for any objects:

Inference Rule XIII

$$\frac{x, w \in RecPair(A)}{pr_r(update_r(x, w, x)) = update(x, w, w)}$$

Finally, if a variable `q` of the type `RecPairA` had value y, different from x, before the evaluation 3.13 took place, and r-component of object y was equal to the value of variable `t`, then after the evaluation r-component of new value of variable `q` is also equal to the new value of variable `t`:

$$pr_r(env'(\mathtt{q})) = env'(\mathtt{t})$$

From this equation, using equality 3.14 we can derive that

$$pr_r(update_r(x, w, y)) = update_r(x, w, pr_r(y))$$

Assuming that this equation holds in Type Theory for any objects $x \neq y$, and w, not only for named by some variables, we can get the following rule:

Inference Rule XIV

$$\frac{x, w, y \in RecPair(A) \qquad x \neq y}{pr_r(update_r(x, w, y)) = update_r(x, w, pr_r(y))}$$

Just as in case of assignment 3.9, we know that assignment 3.13 satisfies the property

$$env'(\mathtt{p}) = env'(\mathtt{q}) \Rightarrow env(\mathtt{p}) = env(\mathtt{q})$$

Therefore, $\lambda y.update_r(x, w, y)$ is a 1-1 mapping:

Inference Rule XV

$$\frac{update_r(x, w, y_1) = update_r(x, w, y_2)}{y_1 = y_2}$$

The rules XII, XIII, XIV, and XV translated into the language of our graph model for type $RecPair(A)$, state that

$$\lambda y.update_r(x, w, y)$$

is a 1-1 endomorphism of type $RecPair(A)$ that preserves vertex labels and all edges except for the edge that starts at vertex x. This edge is mapped into the edge that goes from the image of vertex x to the image of vertex w.

3.1.9 Canonical Elements of Type $RecPair(A)$

After the introduction of *init*, pr_a, pr_r, $update_a$, and $update_r$ there are plenty of elements in the type $RecPair(A)$ that we can define. Basically, any well-typed expression that uses *init*, pr_r, $update_a$, and $update_r$[9] returns an element of type $RecPair(A)$. We will call such elements of type $RecPair(A)$ *definable elements*. There are two questions that probably almost everyone would ask

1. Can two different expressions of the form described above define the same element?

2. Are there any other elements in type $RecPair(A)$ except for definable?

The answer to the first question is simple. Already the rule V gives us an example of two different expressions that are equal as $RecPair(A)$ type elements:

$$pr_r(init(a_0)) = init(a_0)$$

In fact, from rules V, IX, XIII, and XIV it follows that any expression that is built from *init*, pr_a, $update_a$, and $update_r$ can be reduced to an expression that uses only *init*, $update_a$, and $update_r$. An expression that uses only elements of type A and operators *init*, $update_a$, and $update_r$ we will call *canonical expressions*. The values of canonical expressions will be called *canonical elements* of type $RecPair(A)$. As we just have shown, any definable element is a canonical element.

The question whether any two canonical elements are different cannot be solved using existing rules. In fact, various people may be inclined to answer this question differently depending on their intuition and goals. I have found that for Java semantics it is enough to assume that all canonical elements are different. But many participants of PRL Seminar at Cornell argued that this assumption results in the fact that the following two intuitively equivalent Java pieces of code correspond to different changes in the environment:

```
p.a = 5;                                p.a = 7;
p.a = 7;                                p.a = 5;
```

Although there is nothing wrong with the fact that these two programs will be represented in Type Theory by different functions, I would agree that the existence of a stronger but still adequate for our purposes equality on type $RecPair(A)$ is an interesting problem for which I do not have a good solution at the present time.

In this dissertation I will assume that all canonical elements are different. This fact is formalized by the following five rules:

Inference Rule XVI

$$\frac{update_a(x_1, b_1, y_1) = update_a(x_2, b_2, y_2)}{(x_1 = x_2)\ \&\ (b_1 = b_2)}$$

Inference Rule XVII

$$\frac{update_a(x_1, w_1, y_1) = update_a(x_2, w_2, y_2)}{(x_1 = x_2)\ \&\ (w_1 = w_2)}$$

Inference Rule XVIII

$$\frac{a_0, b \in A \qquad x, y \in RecPair(A)}{init(a_0) \neq update_a(x, b, y)}$$

Inference Rule XIX

$$\frac{a_0 \in A \qquad x, w, y \in RecPair(A)}{init(a_0) \neq update_r(x, w, y)}$$

Inference Rule XX

$$\frac{b \in A \qquad x_1, y_1, x_2, w, y_2 \in RecPair(A)}{update_a(x_1, b, y_1) \neq update_r(x_2, w, y_2)}$$

[9] I exclude pr_a from this list because it has type A

The second question is simpler. Since any Java program can be decomposed into a sequence of assignments of different values to object component, it seems reasonable to assume that canonical elements are the only ones that we will need for Java semantics. My implementation of Java semantics in Nuprl proves it.

Therefore, we assume that there are no other elements in $RecPair(A)$ type besides canonical ones. This fact can be formalized as an induction principle for type $RecPair(A)$:

Inference Rule XXI

$$\frac{\begin{array}{c} H;\ a : A;\ J[init(a)/r] \vdash P[init(a)/r] \\ H;\ x, y : RecPair(A);\ b : A;\ J[update_a(x,b,y)/r] \vdash P[update_a(x,b,y)/r] \\ H;\ x, w, y : RecPair(A);\ J[update_r(x,w,y)/r] \vdash P[update_r(x,w,y)/r] \end{array}}{H;\ r : RecPair(A);\ J \vdash P}$$

The induction principle completes the formalization of type $RecPair(A)$ in Type Theory. In the next section I will explain how to use the existing operators to model Java `new` constructor.

3.1.10 Modeling Java Constructor `new`

It is common to associate Java `new RecPairA()` constructor with some kind of magic hat from which one can take a new object each time when an assignment like

$$\texttt{p} = \texttt{new RecPairA}(); \tag{3.15}$$

is evaluated. But there is no magic in Type Theory – each element of every type exists at any time.

To model assignment 3.15 in Type Theory I will use a very different idea. I assume that there is a 1-1 endomorphism h from type $RecPair(A)$ into itself that preserves operations pr_a, pr_r, $update_a$, and $update_r$. For any such mapping h and for any environment env, environment

$$env' = h \circ env$$

would be equivalent to the environment env at least in the sense that they intuitively represent the same state of Java Virtual Machine. I also assume that function h is not a surjection. This means that there is such an element r_0 in type $RecPair(A)$ which is not a member of $h(RecPair(A))$. See Figure 3.5 for an illustration.

If such endomorphism h exists, then Java assignment 3.15 can be represented in Type Theory by a function $update$ that maps an environment env into environment

$$env' = \lambda v. \begin{cases} r_0 & \text{if } v = \texttt{p} \\ h \circ env(v) & \text{otherwise} \end{cases}$$

As one can see, I model Java assignment 3.15 similarly to assignments 3.9 and 3.13. The important difference is that this time we do not need to add a new primitive abstraction, because for any $a_0 \in A$,

$$h := \lambda y. update_r(init(a_0), init(a_0), y)$$

$$r_0 := init(a_0)$$

satisfy all the properties of the function h and of the element r_0 stated above.

3.2 Recursive Types

Reference type $RecPair$ that was introduced above as well as the general notion of reference type that will be discussed in the next section, are the examples of recursively defined types. Recursive types are not new to Type Theory. Another two recursive types, inductive and co-inductive types, have been studied for two decades ([26], [9], [7], [14], [14], [8], [12]). In this section I will show how reference types differ from inductive and co-inductive types and why the last two can not be used in J language semantics instead of reference types.

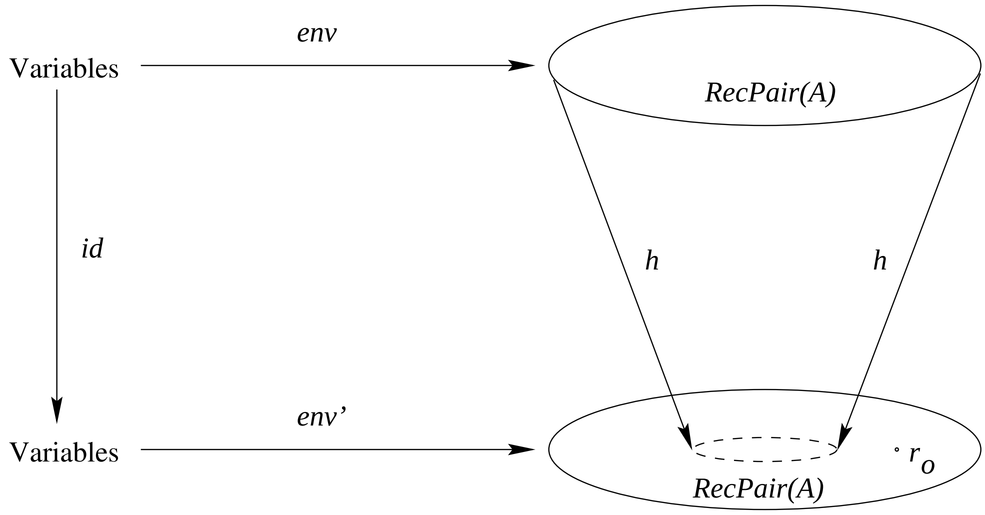

Figure 3.5: Constructor `new` in Type Theory

3.2.1 Inductive type

Inductive type is the minimal solution of the type equation

$$X = B(X)$$

where $B(X)$ is a monotonic, with respect to variable X, expression[10]. The closest approximation to reference type *RecPair* among inductive types is the minimal solution of the following type equation

$$X = \mathbb{Z} \times (X + Unit)$$

where $\mathbb{Z}$ is type of integers, $\times$ stands for Cartesian product, $+$ denotes disjoint union, and $Unit$ is a single-element type. Unfortunately, this inductive type does not contain elements with loops. Hence, not all objects of the J class `RecPair` can be represented by elements of this type.

Also, any two elements of this inductive type that have equal components are equal. This equality is too strong to be used in J semantics, because just like in Java, two different instances of J class `RecPair` can have the same components.

3.2.2 Co-Inductive Types

Co-inductive type is often informally defined as "the maximal" solution of the type equation

$$X = B(X)$$

Such "maximal" solution includes syntactically infinite objects. For instance, co-inductive solution of the equation

$$X = \mathbb{Z} \times X \tag{3.16}$$

includes the element $\langle 3, \langle 1, \langle 4, \langle 1, \langle \ldots \rangle\rangle\rangle\rangle$. Such elements are also called *streams* because of their infinite structure.

Streams from the equation 3.16 co-inductive solution can be used to represent elements of *RecPair* type because any element of *RecPair* type can be "unfolded" into such a stream. For example, *RecPair* element

[10]See page 26 for more detailed discussion of inductive types in Nuprl Type Theory.

init(151) (see Figure 3.2 on page 10) can be represented by the stream $\langle 151, \langle 151, \langle \ldots \rangle\rangle\rangle$. Unfortunately, such representation is not unique in the sense that both elements of type *RecPair*, presented on Figure 3.3 (page 10) are also represented by the same stream $\langle 151, \langle 151, \langle \ldots \rangle\rangle\rangle$. This ambiguity, of course, is closely related to the fact that co-inductive type elements, unlike reference type elements, are uniquely determined my their components.

Therefore, co-inductive types also could not be used to represent J reference types. New type constructor, such as the one proposed in this thesis, should be added to the Type Theory.

3.3 Class Type

3.3.1 Class Signature

The presented above axiomatization of the type *RecPair* is just an example of Java classes formalization in Type Theory. In this section we will look at how the same technique can be applied in a more general case. We will use the term *class type* for referring to new types that we will add to Type Theory to represent Java classes.

Although objects of Java class `RecPairA` have only two components, objects of other Java classes can have different number of components depending on which class they belong to. All components of any Java class can be divided into two categories: *primitive* and *reference* according to their type[11]. For instance, class `RecPairA` has one primitive component and one reference component: a-component and r-component correspondingly. In Type Theory we will use the same terms "primitive" and "reference" for appropriate class type components. From inference rules for $RecPair(A)$ type we already know that primitive and reference components would have different properties.

One of the possible ways to generalize type $RecPair(A)$ is to consider class types that have several primitive and several reference components. But it turns out that Java classes with several primitive components can be modeled by class types with just one primitive component. For example, Java class

```
class X {int a; X x; boolean b;}
```

can be represented in Type Theory by type $RecPair(\mathbb{Z} \times \mathbb{B})$.

Since the same construction cannot be applied to reference components, we will assume that class types may have several reference components, but only one primitive component. Each class type will have *Index* type associated with it. The *Index* type is the type of names for reference components. We will call the unique primitive component of a class type element a *core* of this element. Each class type has some type reserved for cores of its elements. We will call this type *Core*. In the case of $RecPair(A)$ class type, *Index* is any type with only one element and *Core* type is type A.

An important Java feature is the ability to define simultaneously several classes that can use each other as component types. For example,

$$\begin{array}{l} \texttt{X class } \{\texttt{int a; Y y;}\} \\ \texttt{Y class } \{\texttt{boolean b; X x;}\} \end{array} \tag{3.17}$$

To deal with such situations, we will talk about *parametrized class types*. Parametrized class type is a function from some type, that we call *Name* type, into type universe:

$$Name \to \mathbb{U}$$

For instance, the definition 3.17 can be modeled by a function $f : \mathbb{Z}_2 \to \mathbb{U}$ such that $f(0)$ represents class `X` and $f(1)$ represents class `Y`.

Since different classes can have different number of reference components and different *Core* types, we will assume that *Index* and *Core* are also functions of the type $Name \to \mathbb{U}$.

[11] Java specification [11] defines types `int`, `float`, `char`, and `boolean` as primitive and types `class` and `array` as reference.

For the reason that will be explained later, we want to have function $c \in n : Name \to Core(n)$ that returns a representative element $c(n)$ of the type $Core(n)$ for each class name $n \in Name$. In particular, the existence of such a function implies that $Core(n)$ is not empty for any $n \in Name$.

Finally, every reference component of every parametrized class type has some type. I will call the name of this type a *field*. It can be specified by a function that for any $n \in Name$ and any $i \in Index(n)$ returns a $Field(n, i) \in Name$.

Quintuple

$$\mathcal{S} = \langle Name, Core, c, Index, Field \rangle$$

of the type

$$(Name : \mathbb{U}) \times (Core : Name \to \mathbb{U}) \times (n : Name \to Core(n)) \times$$
$$\times (Index : Name \to \mathbb{U}) \times (n : Name \to Index(n) \to Name)$$

that defines class type will be called *class signature*.

3.3.2 The Graph Model

Our graph model of $RecPair(A)$ type can be easily adopted to a more general situation of parametrized class type. We will think about disjoint union

$$\bigsqcup_{n \in Name} \rho(n)$$

as about one big "super graph". Each vertex of this super-graph is labeled by class name $n \in Name$ and a core value of type $Core(n)$. There will be one edge starting at the vertex for each element of $Index(n)$. This edge will be labeled by the corresponding element of type $Index(n)$ and it will point to appropriate reference component of the element. We will assume that the edge marked by $i \in Index(n)$ points to a vertex with class name $Field(n, i)$. On illustrations it is often more convenient to differentiate vertices by shape in order to denote class name. For example, Figure 3.6 shows a fragment of a "super graph" for Java classes

```
Square class {int n, k; Square s; Circle c;}
Circle class {boolean b; Square s; Circle c1, c2;}
```
(3.18)

3.3.3 Type $\rho(\mathcal{S}@n)$

In this section we will introduce inference rules for a parametrized class type defined by a signature:

$$\mathcal{S} = \langle Name, Core, c, Index, Field \rangle$$

As I have mentioned in the previous section, parametrized class type is a function from type $Name$ into type universe $\mathbb{U}$. For any element n of type $Name$ we will denote the value of this function on the argument n as $\rho(\mathcal{S}@n)$.

We assume that class type $\rho(\mathcal{S}@n)$ exists for any signature $\mathcal{S}$:

Inference Rule I [12]

$$\frac{\begin{array}{c} Name \in \mathbb{U} \qquad Core \in Name \to \mathbb{U} \qquad c \in n : Name \to Core(n) \\ Index \in Name \to \mathbb{U} \qquad n \in Name \\ Field \in n : Name \to Index(n) \to Name \end{array}}{\rho(\mathcal{S}@n) \in \mathbb{U}}$$

Until the end of this chapter we will deal with class types defined by some prespecified class signature. Hence, it will be possible to use more compact notation $\rho(n)$ for the type $\rho(\mathcal{S}@n)$. Below I will introduce destructors, constructor, and operations on the elements of the type $\rho(n)$. I start with destructors since constructor *init* will require slight modification before it can be adopted to parametrized classes.

[12] Careful reader may notice that I have not formulated similar formation rule for $RecPair(A)$ type. It was only because $Recpair(A)$ type inference rules were solely meant to be an example and I tried not to divert reader's attention from others, much less trivial, inference rules.

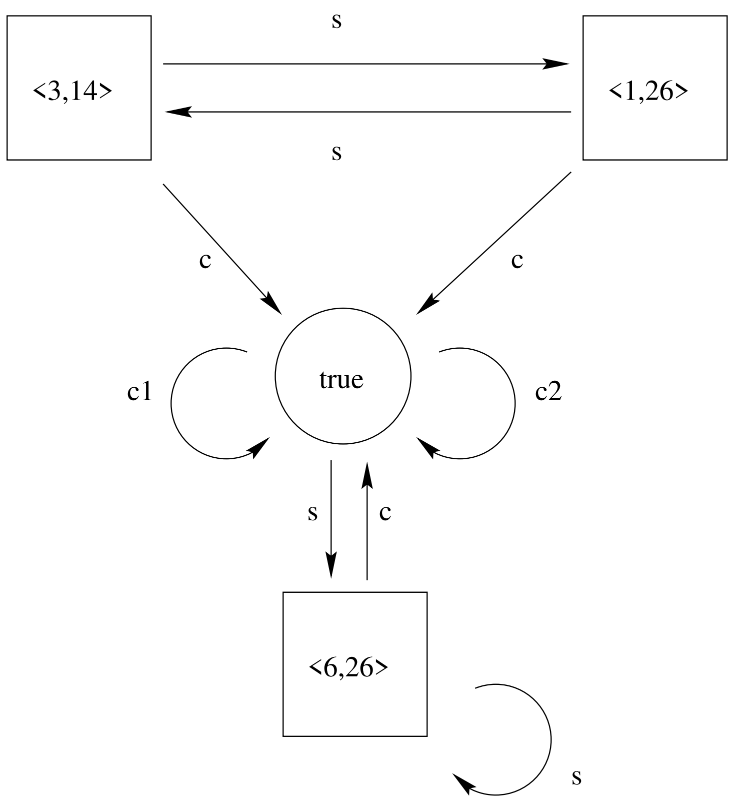

Figure 3.6: Graph model for a parametrized class type

3.3.4 Destructors $core$ and ref

In the case of $RecPair(A)$ type we had two element destructors: projector pr_a returned the core of the element and projector pr_r returned reference component. Since any element of type $\rho(n)$ has one core component and many reference components, the only change that we will need to make is to add reference component name to the second projector. We will call projectors for class type $core$ and ref. The formation rules for them are:

Inference Rule II

$$\frac{x \in \rho(n)}{core(x) \in Core(n)}$$

Inference Rule III

$$\frac{x \in \rho(n) \qquad i \in Index(n)}{ref(x,i) \in \rho(Field(n,i))}$$

3.3.5 Element Constructor $init$

We will need to add class name $n \in Name$ to the argument list of constructor $init$ to specify the type of element that is returned by $init$. Also, as one can see, we will no longer be able to assume that element returned by $init$ constructor has itself as its reference components since the i-th reference component has type $\rho(Field(n,i))$ that may be different from $\rho(n)$.

In my opinion, one of the simplest ways to deal with this ambiguity is to assume that there is only one "initial" element for each class name and any reference component of any initial element is the unique initial element of an appropriate class type:

Inference Rule IV

$$\frac{n \in Name}{init(n) \in \rho(n)}$$

Inference Rule V

$$\frac{n \in Name \qquad i \in Index(n)}{ref(init(n),i) = init(Field(n,i))}$$

Since there would be only one initial element in each class type, we should select some element of type $Core(n)$ that will be the core value of the element $init(n)$. This choice is absolutely unimportant because an operation, similar to $update_a$, will be able to create elements with other core values. We will use signature component c to select such an element of type $Core(n)$:

Inference Rule VI

$$\frac{n \in Name}{core(init(n)) = c(n)}$$

Figure 3.7 displays the relation between the elements $init(`Square`)$ and $init(`Circle`)$ of the class types, corresponding to Java classes 3.18.

3.3.6 Operation $update_c(x,b,y)$

Operation $update_c(x,b,y)$ on type $\rho(n)$ is almost identical to $update_a(x,b,y)$ on the type $RecPair(A)$ except that now x and y can have different types.

Inference Rule VII

$$\frac{x \in \rho(n) \qquad b \in Core(n) \qquad y \in \rho(m)}{update_c(x,b,y) \in \rho(m)}$$

Inference Rule VIII

$$\frac{x \in \rho(n) \qquad update_c(x,b,y) \in \rho(m) \qquad \langle n,x\rangle \neq \langle m,y\rangle}{core(update_c(x,b,y)) = core(y)}$$

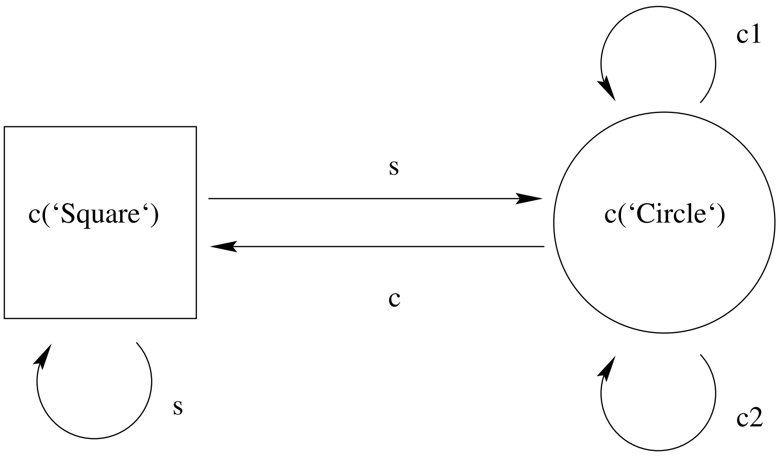

Figure 3.7: Initial Elements

Inference Rule IX

$$\frac{x \in \rho(n) \qquad b \in Core(n)}{core(update_c(x, b, x)) = b}$$

Inference Rule X

$$\frac{update_c(x, b, y) \in \rho(n) \qquad i \in Index(n)}{ref(update_c(x, b, y), i) = update_c(x, b, ref(y, i))}$$

3.3.7 Operation $update_r(x, i, z, y)$

Compared to $update_r$ on $RecPair(A)$ type, reference component update for elements of $\rho(n)$ type has one extra parameter that specifies the name of the reference component being updated:

Inference Rule XI

$$\frac{x \in \rho(n) \qquad i \in Index(n) \qquad z \in \rho(Field(n, i)) \qquad y \in \rho(m)}{update_r(x, i, z, y) \in \rho(m)}$$

Inference Rule XII

$$\frac{update_r(x, i, z, y) \in \rho(n)}{core(update_r(x, i, z, y)) = core(y)}$$

Inference Rule XIII

$$\frac{x \in \rho(n) \qquad update_r(x, i, z, y) \in \rho(m) \qquad j \in Index(m) \qquad \langle n, x, i \rangle \neq \langle m, y, j \rangle}{ref(update_r(x, i, z, y), j) = update_r(x, i, z, ref(y, j))}$$

Inference Rule XIV

$$\frac{update_r(x, i, z, x) \in \rho(n)}{ref(update_r(x, i, z, x), i) = update_r(x, i, z, z)}$$

3.3.8 Canonical Elements

By canonical elements of the type $\rho(n)$ I will mean the minimal subtype $\mathcal{C}(n)$ of the type $\rho(n)$ which satisfies the following three conditions:

1. $init(n) \in \mathcal{C}(n)$ for all $n \in Name$

2. if $x \in \mathcal{C}(n)$, $b \in Core(n)$, and $y \in \mathcal{C}(m)$ then $update_c(x,b,y) \in \mathcal{C}(m)$
3. if $x \in \mathcal{C}(n)$, $i \in Index(n)$, $z \in \rho(Field(n,i))$, and $y \in \mathcal{C}(m)$ then $update_c(x,b,y) \in \mathcal{C}(m)$

All properties of canonical elements formulated for type $RecPair(A)$ will be assumed to hold in the more general case of the class type. Below I just formulate these properties, since we already have discussed them above.

Syntactically different canonical elements are not equal. This fact can be formalized with the following inference rules

Inference Rule XV

$$\frac{n \in Name \qquad update_c(x,b,y) \in \rho(n)}{init(n) \neq update_c(x,b,y)}$$

Inference Rule XVI

$$\frac{n \in Name \qquad update_r(x,i,z,y) \in \rho(n)}{init(n) \neq update_r(x,i,z,y)}$$

Inference Rule XVII

$$\frac{update_c(x_1,b,y_1) \in \rho(n) \qquad update_r(x_2,i,z,y_2) \in \rho(n)}{update_c(x_1,b,y_1) \neq update_r(x_2,i,z,y_2)}$$

Inference Rule XVIII

$$\frac{update_c(x_1,b_1,y_1) = update_c(x_2,b_2,y_2)}{(x_1 = x_2)\ \&\ (b_1 = b_2)\ \&\ (y_1 = y_2)}$$

Inference Rule XIX

$$\frac{update_r(x_1,i_1,z_1,y_1) = update_r(x_2,i_2,z_2,y_2)}{(x_1 = x_2)\ \&\ (i_1 = i_2)\ \&\ (z_1 = z_2)\ \&\ (y_1 = y_2)}$$

Any element of type $\rho(n)$ is canonical. This fact can be stated as induction principle for type $\rho(n)$

Inference Rule XX

$$\frac{\begin{array}{l} H;\ J[init(n)/p] \vdash P[init(n)/p] \\ H;\ m : Name;\ x : \rho(m);\ b : Core(m)\ y : \rho(n); \\ \qquad J[update_c(x,b,y)/p] \vdash P[update_c(x,b,y)/p] \\ H;\ m : Name;\ x : \rho(m);\ i : Index(m);\ z : \rho(Field(m,i)); \\ \qquad y : \rho(n);\ J[update_r(x,i,z,y)/p] \vdash P[update_r(x,i,z,y)/p] \end{array}}{H;\ p : \rho(n);\ J \vdash P}$$

3.3.9 On Modeling `new` constructor

When modeling Java `new` constructor, the procedure described in section on $RecPair(A)$ can be applied in a more general case of class types. Namely, for any class name $n \in Name$, the statement

$$\texttt{p} \ = \ \texttt{new}\,\texttt{n}();$$

corresponds to the the environment update

$$env' = \lambda v. \begin{cases} r_0 & \text{if } v = \texttt{p} \\ h \circ env(v) & \text{otherwise} \end{cases}$$

where

$$r_0 := init(n)$$

$$h := \lambda y.update_c(init(n), c(n), y)$$

Chapter 4

Formalization in Nuprl

4.1 Nuprl Type Theory

This section provides a short and informal sketch of the Nuprl Type Theory. Its purpose is to explain notation that will be used in the rest of this chapter rather than to be an introduction to the Type Theory.

4.1.1 Basic Types

Nuprl Type Theory has several basic type constants: type of integers $\mathbb{Z}$, type of tokens `Atom`, a single-element type `Unit`, and a hierarchy of universes $\mathbb{U}_i$. The unique element of the `Unit` type is denoted by $\cdot$.

Type $\mathbb{Z}$ has the standard set of arithmetical operations: $+$, $-$, and $\times$. Also it is assumed that type $\mathbb{Z}$ is decidable. The last property is formalized by adding to the language the operator

$$\text{if } z_1 = z_2 \text{ then } v \text{ else } w$$

which is equal to v if $z_1 = z_2$ and has value to w otherwise.

Type `Atom` is also assumed to be decidable. Hence, similar operator

$$\text{if } t_1 = t_2 \text{ then } v \text{ else } w$$

is added to the theory.

Types $\mathbb{Z}$, `Atom`, and `Unit` belong to the first type universe $\mathbb{U}_1$ which is a minimal element in the universe hierarchy:

$$\mathbb{U}_1 \in \mathbb{U}_2 \in \mathbb{U}_3 \in \dots$$

Since Nuprl treats propositions as types, the same types $\mathbb{U}_i$ are used as types of propositions. In this case they are commonly written as $\mathbb{P}_i$. Operator *assert* converts any boolean expression b into a proposition $\uparrow b$ that states that this expression is true.

4.1.2 Type Constructors

There are many type constructors in Nuprl Type Theory. Here I will mention only those that will be used later in this chapter.

Cartesian Product

Nuprl uses notation $A \times B$ for the simple Cartesian product of types A and B and notation $a : A \times B$ for dependent Cartesian product. In the case of the dependent Cartesian product, the second type may depend on the choice of the first component. Hence, variable a may occur free in term B.

Element x of the type $a : A \times B$ is constructed by pair constructor $\langle a_0, b_0 \rangle$ and is destructed by projectors $x.1$ and $x.2$. Instead of writing expression $E(x.1, x.2)$ it is more convenient to use shorter notation

$$\text{let } x = \langle a, b \rangle \text{ in } E(x)$$

This construction is a part of Nuprl syntax and is known as the *spread* operator.

Disjoint Union

For any two types A and B, disjoint union of these two types is denoted as $A + B$. Elements of $A + B$ are constructed from the elements of types A and B by constructors $inl(a)$ and $inr(b)$. These elements can be deconstructed using the following case split operator

$$\begin{array}{l}\text{case } x \text{ of} \\ \quad inl(a) \Rightarrow v \\ \quad inr(b) \Rightarrow w\end{array}$$

where variables a and b can occur free in terms v and w correspondingly. For any type A, type $A + Unit$ is often used in Nuprl as a standard single-element "extension" of type A. It has the special display form $?A$.

Function Type

The space of total computable functions from type A into type B is presented in Nuprl by function type $A \to B$. Dependent function type $a : A \to B$ permits the type of the value to depend on argument choice. Elements of the function type, functions, are constructed using standard untyped λ-expressions.

Set Type

For any type T and proposition $P(t)$ over this type, Nuprl Type Theory includes set type {t:T — P(t)}. This set type is a subtype of type T.

Bar Type

In Nuprl Type Theory (see [33], and [6]) partial computable functions from type A into type B are the elements of the type $A \to \overline{B}$ where type $\overline{B}$, also denotable as $bar(B)$, is a type of partial objects that potentially can be evaluated to elements of type B. Unfortunately, existing at this moment Nuprl Bar Type theory is not very well developed because in order to make bar type idea fully compatible with the rest of Nuprl a new *bar-well-formness* lemma[1] should be added to formal library and many basic tactic should be re-written. Until it is done, even a simple proof about bar types requires a lot of work. Such situation with bar types has restricted my verification examples to several very simple programs.

Recursive Type

For any type expression $B(x)$, monotonic with respect to variable X, recursive type $rec(X.B(X))$ is, informally, the minimal solution of the type equation

$$X = B(X)$$

Formal semantics for Nuprl recursive types has been given in [26] by considering the union of the chain

$$X_0 \subseteq X_1 \subseteq X_2 \subseteq \dots$$

in some special Type Theory model, where X_0 is an empty type and $X_{n+1} = B(X_n)$.

Parametrized Recursive Type

In many cases, several types are defined by mutual recursion. For example

$$\begin{cases} X_1 = B_1(X_1, X_2) \\ X_2 = B_2(X_1, X_2) \end{cases}$$

[1] as well as some other lemmata

To handle such types Nuprl uses parametrized recursive type constructor $parec(X, i.B(X, i)@i_0)$. Here X is a function from an index type I into the universe $\mathbb{U}$. The equation system above corresponds to the following expression B:

$$B(X,i) = \begin{cases} \text{if } i = 1 \text{ then } B_1(X(1), X(2)) \\ \text{if } i = 2 \text{ then } B_2(X(1), X(2)) \end{cases}$$

As a part of my work on Java semantics, I have added inference rules for the parametrized recursive types to Nuprl library, since these types are used to define reference types. Parametrized recursive type theory extends standard Type Theory by the following primitive inference rules

parecEquality This rule states the conditions under which the type constructor $parec(X, i.B(X, i)@i_0)$ can be used to create a new type.

```
*R parecEquality
   H ⊢ parec(b1,x1.B1 @ t1) = parec(b2,x2.B2 @ t2) ∈ 𝕌
     BY parecEquality x T b
     H ⊢ t1 = t2 ∈ T
     H, x:T, b:(T → 𝕌) ⊢ B1[b,x/b1,x1] = B2[b,x/b2,x2] ∈ 𝕌'
     H ⊢ Mono{i}(b1,x1.B1 on T)
```

The most non-trivial of these conditions is monotonicity. It states that for any X_1 and X_2 if $X_1(i) \subset X_2(i)$ for any i, then $B(X_1, i) \subset B(X_2, i)$ also for any i:

```
*A mono    Mono{i}(b,x.B[b; x] on T) ==
           ∀b',b'':T → 𝕌.  (∀x:T. b' x ⊆ b'' x)
                   ⇒ (∀x:T. B[b'; x] ⊆ B[b''; x])
```

The monotonicity condition is crucial for the existence of the corresponding parametrized recursive type ([26]). Recursive type equations without monotonicity condition, such as

$$X = \{x \in \mathbb{Z} | \neg(x \in X)\}$$

do not define a recursive type for the obvious reason.

parecMemberEquality This rule specifies elements of parametrized recursive types. Any element of the “unfolded” type also belongs the recursive type. Equality on elements of a recursive type is inherited from the “unfolded” type:

```
*R parecMemberEquality
   H ⊢ b1 = b2  ∈ parec(b,x.B @ t)
       BY parecMemberEquality level{i} z
       H ⊢ b1 = b2 ∈ B[parec(b,x.B @ t),t/b,x]
       H ⊢ parec(b,x.B @ t) ∈ 𝕌
```

parecUnroll The following rule provides logical framework for reasoning about an arbitrary element of the recursive type. Any such element can be thought of as an element of “unfolded” type:

```
*R parecUnroll H, r:parec(b,x.B @ t), J ⊢ G
    BY parecUnroll #$i r' z u
    H, r:parec(b,x.B @ t), J,
       r':B[(λz.parec(b,x.B @ z)),t/b,x], u:(r = r')
       ⊢ G[r'/r]
```

parecElimination The previous rule says that any element of the recursive type is also an element of the "unfolded" type. Since original recursive type is also a part of the "unfolded" expression, it may happen that some element of the recursive type is not a element of any "standard" type. By "standard" type here I mean a type that is not defined in terms of the recursive type constructor. The rule below prohibits such elements of the recursive type.

```
*R parecElimination
  H, t:T, J, r:parec(b,x.B @ t), J1 ⊢ G
     BY parecElimination level{i} #$j1 #$j2 u w v s z
     H, t:T, J, r:parec(b,x.B @ t), J1 ⊢ parec(b,x.B @ t) ∈ 𝕌
     H, t:T, J, r:parec(b,x.B @ t), J1,
        u:(t:T → parec(b,x.B @ t) → 𝕌),
        w:(t:T → r:{v:parec(b,x.B @ t)| u t v} → G), s:T,
        z:B[(λs.{v:parec(b,x.B @ s)| u s v} ),s/b,x]
            ⊢ G[s,z/t,r]
```

In other words, this rule states that $parec(b, x.B@t)$ is *the minimal* solution of the type equation $X(i) = B(X, i)$.

Three inference rules given above constitute a foundation for the theory of the parametrized recursive types. There are some additional, Nuprl-specific, rules in this theory that are not mentioned here[2].

4.2 On Consistency Proofs

In order to add the class type constructor to Nuprl Type Theory we should guarantee the consistency of the new inference rules with the existing ones. The most straightforward way to do it would be to take some model of the existing Nuprl Type Theory and to extend it by a class type constructor in such a way that inference rules for class type hold in extended model. This procedure is a standard technique in the part of Mathematical Logic known as Model Theory because it can be applied to a wide range of different theories. Unfortunately, this approach is also rather complicated, because it requires detailed knowledge of an existing model for Nuprl Type Theory and the understanding of the possible ways of its expansion.

The other disadvantage of the model extension is that consistency proof would be done in a metalanguage outside of Nuprl Proof Development System. Any formalization of this proof would require defining a Type Theory model inside Type Theory itself.

Another way of extending a theory language by new abstractions is more common in theories with expressively rich languages such as Set Theory. In such theories new notions could be *defined* using existing constructions. For example, such important objects as pair, function, natural number, real number, or group can be expressed in Set Theory in terms of sets. In fact, it has been a widely spread belief among mathematicians during the last century that any mathematical object can be defined in Set Theory using only *set* as a primitive notion. Of course, the ability to define all objects in Set Theory using only sets does not depreciate the value of such mathematical theories as Arithmetics, Real Number Theory, or Abstract Algebra. Most mathematicians would agree that although real numbers can be defined in terms of sets, these definitions are only good for providing foundations for Real Number Theory. Available axiomatization of the real numbers makes much more sense for everybody who is studying them.

In this dissertation I will use the second approach. I will define class type using existing constructors in Nuprl Type Theory. These definitions would guarantee the consistency of the extended theory and show expressive power of the existing Type Theory. But just as in the case of Set Theory, these definitions should not be considered to be replacement for class type axiomatization presented in the previous chapter.

4.3 $RecPair(A)$ Type in Nuprl

Just as we did it for the inference rules, let us start with a simple example of the $RecPair(A)$ type that later will be generalized to any class type.

[2]Namely, the rules for dealing with theorem extracts.

The definitions that I give below work for any type A as long as it is decidable. A type is *decidable* if there exists a function $eq_A : A^2 \rightarrow \mathbb{B}$ such that for any elements a and b of type A

$$eq_A(a,b) = true \in \mathbb{B} \Longleftrightarrow a = b \in A$$

Although not all types in Nuprl Type Theory are decidable, the ones that we will use for Java semantics are, so this limitation will not be a problem.

Let $RecPair(A)$ be Nuprl type

$$\mu(X.A + (X \times A \times X) + (X \times X \times X))$$

or, in other words, $RecPair(A)$ is the minimal solution of the following type equation

$$\begin{aligned} RecPair(A) \quad = \quad & A + \\ & + (RecPair(A) \times A \times RecPair(A)) + \\ & + (Recpair(A) \times Recpair(A) \times Recpair(A)) \end{aligned} \tag{4.1}$$

For any decidable type A type $RecPair(A)$, defined this way, is also decidable. Indeed, the function $eq_{RecPair(A)}(r_1, r_2)$ that for any x and y from type $RecPair(A)$ satisfies the condition

$$eq_{Recpair(A)}(x,y) = true \in \mathbb{B} \Longleftrightarrow x = y \in RecPair(A)$$

can be defined by the following recursive procedure

1. If $r_1 = inl(a_1)$ and $r_2 = inl(a_2)$ then return $eq_A(a_1, a_2)$.
2. If $r_1 = inr(inl(\langle x_1, b_1, y_1 \rangle))$ and $r_2 = inr(inl(\langle x_2, b_2, y_2 \rangle))$ then return boolean *true* iff all three of the following expressions have value *true*: $eq_{RecPair(A)}(x_1, x_2)$, $eq_A(b_1, b_2)$, and $eq_{RecPair(A)}(y_1, y_2)$.
3. If $r_1 = inr(inr(\langle x_1, z_1, y_1 \rangle))$ and $r_2 = inr(inr(\langle x_2, z_2, y_2 \rangle))$ then return boolean *true* iff all three of the following expressions have value *true*: $eq_{RecPair(A)}(x_1, x_2)$, $eq_{RecPair(A)}(z_1, z_2)$, and $eq_{RecPair(A)}(y_1, y_2)$.

Function $eq_{Recpair(A)}(x,y)$ will be used later to define projectors pr_a and pr_r on the type $RecPair(A)$.

As one can see from the definition 4.1, there are three ways to construct elements of the type $RecPair(A)$ that correspond to three different components of disjoint union:

$$init(a) = inl(a)$$

$$update_a(x,b,y) = inr(inl(\langle x,b,y \rangle))$$

$$update_r(x,b,y) = inr(inr(\langle x,z,y \rangle))$$

The fact that any element of $RecPair(A)$ type can be constructed from the above defined operations *init*, $update_a$, and $update_r$ justifies the induction principle for type $RecPair(A)$. Also, from 4.1 and basic properties of Cartesian product and disjoint union follows that all canonical elements of the type $RecPair(A)$ are different.

Therefore, we only need to define projectors pr_a and pr_r that would satisfy the inference rules. It can be done if projector $pr_a(p)$ is defined recursively as

1. If $p = init(a)$ then return a.
2. If $p = update_a(x,b,y)$ and $\neg(eq_{RecPair(A)}(x,y))$ then return $pr_a(y)$.
3. If $p = update_a(x,b,y)$ and $eq_{RecPair(A)}(x,y)$ then return b.
4. If $p = update_r(x,z,y)$ then return $pr_a(y)$.

and projector $pr_r(p)$ is defined as

1. If $p = init(a)$ then return p.

2. If $p = update_a(x, b, y)$ then return $update_a(x, b, pr_r(y))$.
3. If $p = update_r(x, z, y)$ and $\neg(eq_{RecPair(A)}(x, y))$
   then return $update_r(x, b, pr_r(y))$.
4. If $p = update_r(x, z, y)$ and $eq_{RecPair(A)}(x, y)$ then return $update_r(x, b, b)$.

Routine check proves that pr_a and pr_r satisfy all the inference rules for type $RecPair(A)$ from the chapter 2.

4.4 Class Type

In the rest of this chapter I will present Nuprl formal theories that contain class type definitions, Java semantics, and examples of Java program verifications. The text of this and of the following sections contains fragments of computer-generated printouts of the formal definitions and theorem statements. Full version of these printouts can be found in the Appendix.

Although reading formal mathematical text is very interesting and quite enjoyable experience that can be only compared to creating such texts, it also may be a difficult task for a person who does it for the first time. In this dissertation I will provide extensive verbal comments for each fragment of the formal text presented here to help to overcome possible difficulties.

The most important fact to remember about these computer printout is that they are using the so called *display forms* of the abstraction that may hide some of its arguments. This is done in Nuprl in order to make formal text more readable, but it also means that these printouts sometimes do not contain sufficient information to reconstruct original definitions or proofs that were stored in the computer memory. Working in the Nuprl Proof Development System a user also usually sees only display forms, but a complete list of parameters for any term can be easily obtained from the system. Dealing with paper version of the proofs one need to use external comments or imagination to understand some pieces of the formal text.

4.4.1 Class Signature

The definition of the class signature in Nuprl differs from the one given in the previous chapter only by three extra functions that make types $Name$, $Core$, and $Index$ decidable. Because types $Core$ and $Index$ are parametrized by names, it will be convenient to have class name of both arguments as a parameters of appropriate eq function. Nuprl definition of a class signature type is

```
*A signature   Sign ==
   nam:𝕌
   × cor:(nam → 𝕌)
   × c:(n:nam → cor n)
   × idx:(nam → 𝕌)
   × fld:(n:nam → idx n → nam)
   × eq_nam:(nam → nam → 𝔹)
   × eq_cor:(n1:nam → cor n1 → n2:nam → cor n2 → 𝔹)
   × (n1:nam → idx n1 → n2:nam → idx n2 → 𝔹)
```

Therefore, class signature is any octuple of the defined above type. This gives us an example of using Nuprl display forms. First of all, although the name of the abstraction is `signature`, here and everywhere below it will be displayed as `Sign` to save space in formulas. Also, `Sign` type has one hidden parameter - universe level `i` which comes from type 𝕌. We will use special name for each component of the octuple `s`:

```
*A sign_nam                Nam ==  s.1
*A sign_cor                Cor ==  s.2.1
*A sign_c                  cor ==  s.2.2.1
*A sign_idx                Idx ==  s.2.2.2.1
*A sign_fld                Fld ==  s.2.2.2.2.1
```

```
*A sign_eq_nam              EqNam ==  s.2.2.2.2.2.1
*A sign_eq_cor              EqCor ==  s.2.2.2.2.2.2.1
*A sign_eq_idx              EqIdx ==  s.2.2.2.2.2.2.2
```

Each of the definitions above has `s` as a hidden parameter. Expression `EqCor n1 c1 n2 c2` just says that elements `c1` and `c2` are equal, but since it has two extra arguments for class names of `c1` and `c2`, it looks too complicated. Unfortunately, we can not hide arguments `n1` and `n2` in this expression because they are arguments of `apply` term, not `sign_eq_cor`, but we can define new abstraction `eq_cor` that will combine `apply` and `sign_eq_cor` together and after this hide arguments `n1` and `n2`:

```
*A eq_cor                    (c1 =c c2) ==  EqCor n1 c1 n2 c2
```

The same thing can be done with `sign_eq_idx`:

```
*A eq_idx                    (i1 =c i2) ==  EqIdx n1 i1 n2 i2
```

In spite of the fact that `sign_eq_nam` does not have extra parameters, I decided to apply the same procedure just to have similar notation and display form:

```
*A eq_nam                    (n1 =c n2) ==  EqNam n1 n2
```

Finally, we are only interested in signatures with *eq* functions that are consistent with equality predicates on appropriate types. Such signatures will be called *regular*:

```
*A reg_sign    Reg(s) ==
   (∀n,k:Nam.  ↑(n =c k) ⟺ n = k)
   ∧ (∀n,k:Nam. ∀c:Cor n. ∀d:Cor k.
              ↑(c =c d) ⟺ (n = k) c∧ (c = d))
   ∧ (∀n,k:Nam. ∀i:Idx n. ∀j:Idx k.
              ↑(i =c j) ⟺ (n = k) c∧ (i = j))
```

In addition to abstractions listed above, theory `signature` also contains corresponding display forms and well-formness theorems. This theory also includes the object `create_signature` that was used by Nuprl module tactic to simplify creation of this theory. It can be removed from the theory at any time.

4.4.2 Class Definition

In Nuprl theory `class`, I define embedding of the class type into existing Nuprl Type Theory. Similarly to the case of type *RecPair*, it can be done by defining class type $\rho(\mathcal{S}@n)$ as

```
*A class       ρ@n ==
   parec(C,j.{k:Nam| k = j}  +  k:Nam × C k × Cor k × C j  +
              k:Nam × C k × i:Idx k × C (Fld k i) × C j @ n)
```

This abstraction has two arguments: signature `s` and class name `n`. Because in most cases one deals with some given signature, the first argument is hidden.

The definition above shows that any class is a disjoint union of three different types. Hence, there are three natural ways to construct elements of a class type. They correspond to operations *init*, $update_c$, and $update_r$. At the moment when library was created, I used the name *nil object* for *init* constructor. Although later I have switched to the name *init* to avoid collision with Java *nil* objects, it was too hard to make appropriate changes in the formal theory. Display form for `nil_object(k)` is `⊕(k)`. Operations $update_c$ and $update_r$ also have fancy display forms:

```
*A nil_object     ⊕(k) ==  inl k
*A cor_update     [core(x:ρ@k):=t@y] ==  inr (inl <k, x, t, y>)
*A ref_update     [(x:ρ@k).i:=z@y] ==  inr inr <k, x, i, z, y>
```

To finish with class type embedding in Nuprl Type Theory I only need to define projector operators on elements of class `ρ@k`. To do it the most elegantly we need several auxiliary abstractions. The first of them is a scheme for definitions by induction. In order to define any function on type `ρ@k` it is enough to define it on elements of the form `⊕(k)`, `[core(x:ρ@k):=t@y]`, and `[(x:ρ@k).i:=z@y]`. Such definitions can be given in Nuprl using the following abstraction:

```
*A object_cases
case o
   of ⊕(n) -> base_case[n]
   | [core(x:ρ@k):=t@y] -> cor_case[k; x; t; y]
   | [(u:ρ@m).i:=v@w] -> ref_case[m; u; i; v; w]
) ==
   case o
      of inl(n) => base_case[n]
      | inr(cr) => case cr
                      of inl(c) => let <k,xty> = c
                                   in
                                   let <x,ty> = xty
                                   in
                                   let <t,y> = ty
                                   in
                                   cor_case[k; x; t; y]
                      | inr(r) => let <m,uivw> = r
                                   in
                                   let <u,ivw> = uivw
                                   in
                                   let <i,vw> = ivw
                                   in
                                   let <v,w> = vw
                                   in
                                   ref_case[m; u; i; v; w]
```

In spite of its long definition, abstraction `object_cases` has very simple meaning. It takes any element `o` of type `ρ@k` and returns

1. `base_case[n]` if `o` has form `inl(n)` for some `n`
2. `cor_case[k; x; t; y]` if `o` has form `inl(inr <k,x,t,y>)`
3. `ref_case[m; u; i; v; w]` if `o` has form `inr(inr <m,u,i,v,w>)`

By "has form" above I mean that `o` is a member of appropriate type in disjoint union `ρ@k`. If the term `o` *syntactically* has one of the forms above then expression `object_cases` can be reduced to `base_case[n]`, `cor_case[k; x; t; y]`, or `ref_case[m; u; i; v; w]`. Such reductions can be handled automatically by Nuprl `Reduce` tactic. Library object `object_cases_eval` (see complete theory printout in the Appendix) extends `Reduce` tactic in an appropriate way.

In addition to *definitions* by induction on type `ρ@k`, we also will use *proofs* by induction on this type. In many instances such proofs can be produced by applying the following lemma[3]

```
*T class_ind_tp
∀s:Sign. ∀P:n:Nam → ρ@n → ℙ.
   (∀n,k:Nam.  k = n ⇒ P[n;⊕(k)])
   ⇒ (∀n,k:Nam. ∀x:ρ@k. ∀t:Cor k. ∀y:ρ@n.
```

[3]Formal proof printouts of this and any other theorem mentioned in this chapter can be found in the Appendix.

```
      P[k;x] ⇒ P[n;y] ⇒ P[n;[core(x:ρ@k):=t@y]])
   ⇒ (∀n,k:Nam. ∀x:ρ@k. ∀i:Idx k. ∀z:ρ@Fld k i. ∀y:ρ@n.
      P[k;x] ⇒ P[Fld k i;z] ⇒ P[n;y]
                               ⇒ P[n;[(x:ρ@k).i:=z@y]])
   ⇒ {∀n:Nam. ∀o:ρ@n.  P[n;o]}
```

But in some cases, especially in well-formness theorem proofs, we can not apply this lemma because we do not know *a priori* whether `P` has type `n:Nam` $\to$ `ρ@n` $\to$ $\mathbb{P}$. In such situations we can just try to repeat the *proof* of the theorem above. It is done by the tactic `ClassInd`:

```
*M class_ind_tac
   let ClassInd i j p = ByAnalogyWith1 `class_ind_tp`
                      [`hr_n`, int_to_arg i;
                       `hr_o`, int_to_arg j] p
   ;;
```

Of course, the proof of `class_ind_tp` theorem was written to be general enough so that it could be "re-used" on similar goals.

Using abstraction `object_cases` we can define recursive function `nam` that returns the class name of any class type element

```
*M nam_ml
  nam(o) ==r case o
              of ⊕(k) -> k
               | [core(x:ρ@k):=t@y] -> nam(y)
               | [(x:ρ@k).i:=z@y] -> nam(y)
```

and boolean relation `eq_obj` between elements of (possibly different) class types. This relation has boolean value `true` only if the elements belong to the same class and are equal to each other:

```
*M eq_obj_ml   (o1 = o o2)
==r case o1
       of ⊕(k1) ->
          case o2
             of ⊕(k2) -> (k1 =c k2)
             | [core(x2:ρ@k2):=t2@y2] -> ff
             | [(x2:ρ@k2).i2:=z2@y2] -> ff
       | [core(x1:ρ@k1):=t1@y1] ->
          case o2
             of ⊕(k2) -> ff
             | [core(x2:ρ@k2):=t2@y2] -> (k1 =c k2)
                   ∧b (x1 =o x2)
                   ∧b (t1 =c t2)
                   ∧b (y1 =o y2)
             | [(x2:ρ@k2).i2:=z2@y2] -> ff
       | [(x1:ρ@k1).i1:=z1@y1] ->
          case o2
             of ⊕(k2) -> ff
             | [core(x2:ρ@k2):=t2@y2] -> ff
             | [(x2:ρ@k2).i2:=z2@y2] -> (k1 =c k2)
                   ∧b (x1 =o x2)
                   ∧b (i1 =c i2)
                   ∧b (z1 =o z2)
                   ∧b (y1 =o y2)
```

Projectors `cor` and `ref` could be define recursively using `object_cases` and `eq_obj`:

```
*M cor_ml
   cor(o) ==r case o
                 of ⊕(n) -> cor n
                 | [core(x:ρ@k):=t@y] ->
                    if ( x= _0y) then
                       then t
                       else cor(y)
                    fi
                 | [(x:ρ@k).i:=z@y] -> cor(y)
*M ref_ml
   o.i ==r case o
             of ⊕(k) -> ⊕(Fld k i)
             | [core(x:ρ@k):=t@y] -> [core(x:ρ@k):=t@y.i]
             | [(x:ρ@k).j:=z@y] ->
                if (x = _0y) ∧_b (j =_c i)
                   then [(x:ρ@k).j:=z@z]
                   else [(x:ρ@k).j:=z@y.i]
                fi
```

These two definitions complete the formalization of class types in Nuprl Type Theory. But theory `class` has several other objects that I have not mentioned yet. Most of them are lemmata about class type elements that are useful in proofs. There are also three new abstractions:

```
*A c_update      [cor(x):=t@y] ==  [core(x:ρ@nam(x)):=t@y]
*A r_update      [x.i:=z@y] ==  [(x:ρ@nam(x)).i:=z@y]
*A get_ref       get_ref(o;i;s) ==  o.i
```

that will be used instead of `cor_update`, `ref_update`, and `ref` because "element class name" argument of the last three functions is redundant - it can be computed by function `nam` from the element itself.

4.5 J Class Type

In this section I will show how the general concept of class types can be used to formalize in Nuprl Type Theory some of Java types. Since the fragment of Java that will be formalized has been called above J language, we will talk about formalizing *J type structure* in Nuprl.

4.5.1 J Types

Any program on J specifies the list of names of J classes that it uses. Class names in J are selected from tokens. Due to the fact that tokens are represented in Nuprl Type Theory by `Atom` type, J class name list may be associated in Type Theory with a boolean function `f:Atom` $\rightarrow \mathbb{B}$. For any such function `f` we define J class names

```
*A j_class_name
               JClassName(f) ==  {a:Atom| ↑(f a)}
```

After J class names are specified, all possible J types that match this specification can be described as

1. **Primitive types.** In J language we have only three primitive types: `boolean`, `int`, and `Exception`[4].
2. **Classes.** Any element of `JClassName(f)` represents a class.
3. **Arrays.** For any J type `T` there is "the array of `T`" type that is denoted by `T[]`.

[4]Let me remind that although in Java class `Exception` is a reference type that is contained in Standard Class Library, in J we call this type primitive.

Using Nuprl inductive types, the universe of J types for any given function `f` can be defined as

```
*A type        Type ==
                 rec(T.Unit + Unit + Unit + JClassName(f) + T)
```

Since we normally will consider types specified by the same fixed function `f`, argument `f` in abstraction `Type` is hidden. The definition above specifies the universe of J types as some Nuprl type. We will use the following notations for referring to specific J types:

```
*A bool_type                  boolean ==  inl ·
*A int_type                   int ==  inr (inl · )
*A exc_type                   Exception ==  inr inr (inl · )
*A class_type                 Class(n) ==  inr inr inr (inl n )
*A array_type                 t[] ==  inr inr inr inr t
```

As with many other abstractions based on Nuprl recursive types, there are several auxiliary objects that should be added to the library to make proofs and definitions more elegant. We have already done so for `class` types. In the case of `type` abstraction, I add a `type_hd` ML object that contains tactic for hypothesis decomposition, abstraction `type_cases`

```
*A type_cases  case x
               of bool -> b
                | int -> z
                | Exception -> e
                | Class(n) -> c[n]
                | t[] -> a[t]
                ==
                 case x
                 of inl(x1) => b
                  | inr(x2) => case x2
                               of inl(x3) => z
                                | inr(x4) => case x4
                                             of inl(x5) => e
                                              | inr(x6) => cas
               e x6
                                                          of
               inl(n) => c[n]
                                                           |
               inr(t) => a[t]
```

for definitions by induction over the type `Type`, ML object `type_cases_eval` to make `Reduce` tactic to work with `type_cases`, and ML object `type_ind` that contains tactic `TypeInd` for proofs by induction over type `Type`.

In addition, I prove that type `Type` is decidable by constructing boolean relation `eq_type`

```
*M eq_type_ml  (x =t y)
               ==r case x
                   of bool -> case y
                              of bool -> tt
                               | int -> ff
                               | Exception -> ff
                               | Class(n) -> ff
                               | t[] -> ff
                    | int -> case y
                             of bool -> ff
                              | int -> tt
```

```
                | Exception -> ff
                | Class(n) -> ff
                | t[] -> ff
     | Exception -> case y
                    of bool -> ff
                     | int -> ff
                     | Exception -> tt
                     | Class(n) -> ff
                     | t[] -> ff
     | Class(n) -> case y
                   of bool -> ff
                    | int -> ff
                    | Exception -> ff
                    | Class(m) -> n =a m
                    | t[] -> ff
     | t[] -> case y
              of bool -> ff
               | int -> ff
               | Exception -> ff
               | Class(n) -> ff
               | s[] -> (t =t s)
```

on type `Type` that corresponds to equality on this type:

```
*T eq_type_iff_eq
⊢ ∀f:Atom → 𝔹. ∀x,y:Type.  ↑(x =t y) ⟺ x = y
```

Finally, the function `f`, that defines `JClassName(f)`, does not define class structure completely, because we also should specify class fields and their types. In my formalization it is done by a function of type

```
*A spec        Spec ==  JClassName(f) → Atom → ?Type
```

Any function of this type for any class name and field name returns `(inr ·)`, if corresponding J class does not have such field name, otherwise it returns `(inl t)` where element `t` of type `Type` represents J type of the field.

A function `f: Atom` → 𝔹 combined with the specification `s` of type `Spec` defines the abstract data structure for a J program. As a next step I will show how to build a class signature corresponding to any such structure.

4.5.2 J Class Signature

First I want to discuss two major differences between J abstract data structures and Nuprl class types.

- **Dealing with J primitive types.** I will represent J primitive types `boolean`, `int`, and `Exception` as class types with an empty set of reference components, not as standard Nuprl type $\mathbb{B}$, $\mathbb{Z}$, and **Atom**. It will help to keep definitions more uniform. The major difference between, say, $\mathbb{B}$ type and class type with $Core = \mathbb{B}$ and empty set of reference components is the element equation. There are infinitely many different class type elements that have the same *core* value. Since J type `boolean` has only two unequal elements, we will need to represent J equality on type `boolean` as equality of core values instead of just equality of elements.

- **Dealing with nil pointers.** Before any value is assigned to a J variable or J object field, it has some special *nil* value of the appropriate type. I will model it in Type Theory by adding special nil value to $Core(n)$ type for each class name n:

$$Core(n) = ActiveCore(n) + Unit$$

where *ActiveCore*(n) is a type of core values for real elements of the class type and a unique element of *Unit* type corresponds to nil pointer of an appropriate type. It is important to note that normally Java compiler will not accept a program that operates with nil pointers. Hence real Java program will not refer to any variable or object field before some value is assigned to it. Unfortunately, formalizing such notion of program is rather difficult, instead I will consider broader class of programs that can use unassigned variables or fields.

Now we are ready to define a class signature for any J abstract data structure. Since we have agreed to represent all J types as a class types, there will be exactly one class type for each element of `Type` type. I will use type `Type` as class name type *Name*. For any element `x` of type `Type`, active core type `JActCor(x)` can be defined using `type_cases` abstraction:

```
*A j_act_cor   JActCor(x) ==
                 case x
                 of bool -> 𝔹
                  | int -> ℤ
                  | Exception -> Atom
                  | Class(n) -> Void
                  | t[] -> ℕ
```

This means that for J types `boolean`, `int`, and `Exception` the value of `JActCor` are $\mathbb{B}$, $\mathbb{Z}$, and `Atom` correspondingly. Since all components of J class type are reference components, active core type of any `Class(n)` is empty. J array type will be represented as a class type the core element of which stores the length of the array.

For any `x` of type `Type`, `JActCor(x)` is a decidable type. Boolean equality function can be defined on it as

```
*A j_eq_act_cor
               (a =ac b) ==
                 case x
                 of bool -> a =b b
                  | int -> (a =z b)
                  | Exception -> a =a b
                  | Class(n) -> tt
                  | t[] -> (a =z b)
```

As has been mentioned above, core type is a disjoint union of the active core type and the single-element type *Unit*:

```
*A j_cor                      JCor(x) ==  ?JActCor(x)
```

I will use two abstractions to construct elements of `JCor(x)` type from different components of the disjoint union

```
*A j_c                        Jc ==  inr ·
*A j_ac                       Jac(a) ==  inl a
```

and the abstraction `j_cor_cases` to define functions on `JCor(x)` type:

```
*A j_cor_cases case x of Jc -> c  | Jac(a) -> ac[a] ==
                 case x of inl(a) => ac[a] | inr(i) => c
```

In order for type `JCor(x)` to be `Cor` type in a class signature it should be decidable. Boolean equality can be defined on `JCor(x)` using `j_cor_cases`:

```
*A j_eq_cor    (a =c b) ==
                 case a
                 of Jc -> case b of Jc -> tt  | Jac(u) -> ff
```

```
                | Jac(ac) -> case b
                              of Jc -> ff
                               | Jac(bc) -> (ac =ac bc)
```

The following theorem proves that `j_eq_cor` is indeed a boolean equality:

```
*T j_eq_cor_iff_eq
⊢ ∀f:Atom → 𝔹. ∀x:Type. ∀a,b:JCor(x).  ↑(a =c b) ⟺ a = b
```

Before defining *Index* type for a J abstract data structure I need to explain how arrays are represented using the class type notations. Since Java array type does not specify array length, arrays of different length belong to the same class. Hence, in order to view array type as a Nuprl class type we will assume that array type has infinitely many reference components. The real length of the array is stored as its core value. This approach works, although it requires boundary checking each time when an array element is accessed. I find it acceptable because Java Virtual Machine employs the same procedure.

Therefore, *Index* type for any class name `x` and any class specification `s` can be defined as

```
*A j_idx       JIdx(x) ==
                 case x
                 of bool -> Void
                  | int -> Void
                  | Exception -> Void
                  | Class(n) -> {a:Atom| ↑isl(s n a)}
                  | t[] -> ℕ
```

The decidability of the type `JIdx(x)` for any `x` from type `Type` is guaranteed by the following boolean equality function

```
*A j_eq_idx    (i =i j) ==
                 case x
                 of bool -> tt
                  | int -> tt
                  | Exception -> tt
                  | Class(n) -> i =a j
                  | t[] -> (i =z j)
```

and by the corresponding theorem

```
*T j_eq_idx_iff_eq
⊢ ∀f:Atom → 𝔹. ∀x:Type. ∀s:Spec. ∀i,j:JIdx(x).
     ↑(i =i j) ⟺ i = j
```

Lastly, we should give the definition of a *Field* function in order to specify class signature for a J abstract data structure.

```
*A j_fld       Jfld(x.i) ==
                 case x
                 of bool -> boolean
                  | int -> int
                  | Exception -> Exception
                  | Class(n) -> outl(s n i)
                  | t[] -> t
```

At the first glance this definition seems to contradict the reader's expectations. Previously I mentioned that class types for `boolean`, `int`, and `Exception` will have no reference components and now I am saying that the types of these component are `boolean`, `int`, and `Exception` correspondingly. There is no contradiction here because we are allowed to assume anything about a non-existing object. The way Nuprl logic is built, there should be some expression in any branch of case split even if we can prove that it never will be used.

I decided to put `boolean`, `int`, and `Exception`, but it was possible to put any other (maybe even equal) elements of type `Type`.

Using introduced above notations we can define J class signature as

```
*A j_sign      JSign(f;s) ==
                 <Type
                 , λt.JCor(t)
                 , λt.Jc
                 , λt.JIdx(t)
                 , λt,i.Jfld(t.i)
                 , λt1,t2.(t1 =t t2)
                 , λt1,c1,t2,c2.
                    if (t1 =t t2) then (c1 =c c2) else ff fi
                 , λt1,i1,t2,i2.
                    if (t1 =t t2) then (i1 =i i2) else ff fi >
```

Simple combination of the stated above theorems about boolean equality functions proves the that this class signature is regular:

```
*T j_sign_reg
⊢ ∀f:Atom → 𝔹. ∀s:Spec.  Reg(JSign(f;s))
```

4.5.3 J Class Definition

For any function `f: Atom` $\to \mathbb{B}$ and any specification `s` we can define J signature `JSign(f;s)`. Parametrized class defined by this signature will be called `j_class`

```
*A j_class                    J(t) ==  ρ@t
```

Any J object of type t will be modeled in our semantics by some element of class type `J(t)`, where `t` is an element of type `Type` that corresponds to J type t. Although general class type theory provides abstractions for constructors of the type `J(t)` elements as well and defined the operations on these elements, in the theory `j_class` I introduce new notations for them. The main reason for this is to simplify type checking proofs by Nuprl `Auto` tactic.

In the special case of type `J(t)` constructors `nil_object`, `c_update`, and `r_update` will be called `j_nil_obj`, `j_c_update`, and `j_r_update`:

```
*A j_nil_obj                  t0 ==  ⊕(t)
*A j_c_update                 [cor(x):=c@y] ==  [cor(x):=c@y]
*A j_r_update                 [x.i:=z@y] ==  [x.i:=z@y]
```

Similarly, destructors `cor`, `ref`, and `nam` will be called `get_j_cor`, `get_j_ref`, and `get_type`:

```
*A get_j_cor                  get_j_cor(x;f;s) ==  cor(x)
*A get_j_ref                  (x.i) ==  get_ref(x;i;JSign(f;s))
*A get_type                    get_type(x) ==  nam(x)
```

and boolean equality `eq_obj` is known as `eq_j`

```
*A eq_j                       eq_j(x;y;f;s) ==  (x=0y)
```

The following constructor returns an element of type `J(t)` with core value `c` for any element `t` of type `Type` and element `c` of type `JCor(t)`:

```
*A j_cor_const JCorConst(c;t) ==  [cor(t0):=c@t0]
```

We also will often use constructor

```
*A j_const     JConst(c;t) ==  JCorConst(Jac(c);t)
```

the argument `c` of which is a member of `JActCor(t)` type.

Let me remind that J types `int` and `boolean` will be modeled as class types `J(int)` and `J(boolean)` that have empty list of reference fields. There are many standard binary operations on elements of these types: addition, subtraction, multiplication, boolean *and*, boolean *or*, etc. Theory `j_class` specifies a scheme to define these operations on types `J(int)` and `J(boolean)` using corresponding operations on Nuprl types $\mathbb{Z}$ and $\mathbb{B}$. It is done in two steps. First, any operation on `JActCor(t)` type is extended to be an operation on `JCor(t)` type:

```
*A j_cor_oper2 j_cor_oper2(u,v.op[u; v];x;y) ==
                 case x
                 of Jc -> Jc
                  | Jac(ax) -> case y
                               of Jc -> Jc
                                | Jac(ay) -> Jac(op[ax; ay])
```

and after that it is converted into operation on `J(t)` type:

```
*A simple_obj_oper2
               simple_obj_oper2(u,v.op[u; v];x;y;f;s;t) ==
                 JCorConst(j_cor_oper2(u,v.op[u; v];get_j_cor(
               x;f;s);get_j_cor(x;f;s));t)
```

4.6 J Semantics

This section applies Reference Type Theory to describe denotational semantics of the J language based on shallow embedding of the J into Reference Type Theory. Useage of the shallow embedding eliminates the need to formalize J syntax inside Type Theory.

4.6.1 Environment

The current state of the J Virtual Machine is completely described in our semantics by the environment, where environment is the mapping from variable names into variable values. We will represent J variables by tokens. Tokens in Nuprl Type Theory are elements of `Atom` type. Hence, environment is a function of type `Atom` $\rightarrow$`J(t)` for every `t` of type `Type`. One of the possible ways to avoid parametrization of the environment by types is to define it as

```
*A env                        Env ==  Atom → t:Type → J(t)
```

As one can see, this definition is so general that it technically permits to have several variables under the same name as long as they are declared to have different types[5].

Theory `env` also introduces abstraction `env_update` that changes the value of the environment function for some argument

```
*A env_update  en{(v:tv) -> o} ==
                 λw,tw.
                  if v =a w ∧b (tv =t tw)
                  then o
                  else en w tw
                  fi
```

[5]In Java the same variable name cannot be used to denote variables of different types.

4.6.2 Expression

An expression can be thought of as a function that for any environment returns the result and a new environment. The new environment can be different from the original one because evaluation of a Java expression may create a side effect. The result that is returned by an expression does not necessarily have the same type as the original expression. In the case if evaluation throws an exception, the result will have type `Exception`. In type theory we will assume that the result is a disjoint union of the exception type and the expression type.

Therefore, value of an expression in Java can be represented by the following type

```
*A exp_value   ExpValue(t) ==  Env × (J(Exception) + J(t))
```

There are two constructors for type `ExpValue(t)` that are defined in Nuprl theory `exp`. They correspond to abrupt and normal expression evaluation termination:

```
*A exp_value_exc
               ExpValueExc(en;ex) ==  <en, inl ex >
*A exp_value_make
               ExpValueMake(en;x) ==  <en, inr x >
```

I also define abstraction `exp_value_cases` that will be useful to define functions on `ExpValue(t)` type:

```
*A exp_value_cases
               case ev
               of Exc(en1,ex) -> exc[en1; ex]
                | Make(en2,x) -> mk[en2; x]
                ==
                 let <en,et> = ev
                 in
                 case et
                 of inl(ex) => exc[en; ex]
                  | inr(x) => mk[en; x]
```

As was mentioned above, any Java expression is a function that maps the environment into `ExpValue(t)`. Since some expression evaluations never terminate, this is a partial function. Nuprl deals with partial functions using bar type. Accordingly, an expression of type `t` is a function from environment type into bar over the expression value type:

```
*A exp         Exp(t) ==  Env → bar(ExpValue(t))
```

In the rest of `exp` theory I define semantics for some sample Java expressions. In many cases different Java expressions are represented by almost identical type theory expressions. In such cases I first define the general scheme for such expressions. For example, I first specify the general notion of a constant expression of type `t`:

```
*A const_exp   const_exp(c;t) ==
                 λen.ExpValueMake(en;JConst(c;t))
```

and after this use this abstraction to define constants of types `int`, `boolean`, and `Exception`:

```
*A int_const                  (z) ==  const_exp(z;int)
*A bool_const                 (b) ==  const_exp(b;boolean)
*A exc_const                  (a) ==  const_exp(a;Exception)
```

In Java, constant integer expression with value 5 is denoted by `5`. I use display form `(5)` to distinguish it from the integer number 5. Appropriate display form can always be changed to match Java syntax.

Another example of a general scheme for Java expression semantics is `simple_exp2`

```
*A simple_exp2 simple_exp2(u,v.op[u; v];x;y;f;s;t) ==
                 λen.case x en
                     of Exc(en',ex) -> ExpValueExc(en';ex)
                      | Make(en',vx) -> case y en'
                                        of Exc(en'',ex) ->
               ExpValueExc(en'';ex)
                                         | Make(en'',vy) ->
               ExpValueMake(en'';simple_obj_oper2(u,v.op[u; v];v
               x;vy;f;s;t))
```

that can be used to define many standard binary operations on integers and booleans:

```
*A add_exp     (x + y) ==  simple_exp2(u,v.u + v;x;y;f;s;int)
*A sub_exp     (x - y) ==  simple_exp2(u,v.u - v;x;y;f;s;int)
*A mul_exp     (x × y) ==  simple_exp2(u,v.u * v;x;y;f;s;int)
*A and_exp     (x & y) ==
                 simple_exp2(u,v.u ∧b v;x;y;f;s;boolean)
```

Theory exp also contains some other examples of semantics for Java expressions. Since describing semantics for all Java expressions involves nothing else than routine application of the described above ideas, I have not persuaded this goal.

4.6.3 Statement

Although Java language specification [11] considers statements as expressions, it is more convenient to have independent statement definition as a partial function from the environment into the statement value type:

```
*A stmt_value  StmtValue(f;s) ==  Env × ?J(Exception)
```

It means that the statement as a function returns a new environment and either an exception, if it terminates abruptly, or the unique element of Unit type, if it terminates normally. Similarly to type ExpValue(t), type StmtValue has two constructors:

```
*A stmt_value_exc
               StmtValueExc(en;ex) ==  <en, inl ex >
*A stmt_value_norm
               StmtValueNorm(en) ==  <en, inr · >
```

and a scheme for definition by case split is as follows:

```
*A stmt_value_cases
               case stv
               of StmtValueExc(en,ex) -> E[en; ex]
                | StmtValueNorm(en) -> N[en]
                ==
                 let <en,r> = stv
                 in
                 case r
                 of inl(ex) => E[en; ex]
                  | inr(i) => N[en]
```

The type of statements can be defined using bar type constructor:

```
*A stmt        Stmt ==  Env → bar(StmtValue(f;s))
```

Theory stmt also provides examples how standard Java statements can be specified as functions of the type Stmt. Among them are:

```
*A throw       throw(e) ==
                 λen.case e en
                     of Exc(en',ex) -> StmtValueExc(en';ex)
                      | Make(en',v) -> StmtValueExc(en';v)
*A if          if(b;c;d;f;s) ==
                 λen.case b en
                     of Exc(en',ex) -> StmtValueExc(en';ex)
                      | Make(en',bv) -> case get_j_cor(bv;f;s)
*A var_assig   (v:t) := e ==
                 λen.case e en
                     of Exc(en',exc) -> StmtValueExc(en';exc)
                      | Make(en',o) -> StmtValueNorm(en'{(v:t)
                -> o})
```

4.7 Verification

Although the primary topic of this dissertation is the description of a Java language semantics and a reference object theory that provides the foundation for this semantics, I have also applied the semantics to program verification. Since I have only tried to explore possible applications for my semantics, I have not verified any big program. Instead, my main focus was on developing appropriate verification technique.

The most obvious way to verify a program in a formal denotational semantics is to directly prove theorems about the function representing this program in Type Theory. Unfortunately, such function is normally too complicated to prove any theorem about it without a well-designed methodology.

In my experience, Hoare logic can be successfully used as such methodology. Hoare ([15]) introduced predicate $\{P\}c\{Q\}$ also know as Hoare triple[6]. Predicate $\{P\}c\{Q\}$ states that if the pre-condition P on the state of an abstract machine is true before program c is executed, then the post-condition Q is true after the execution. In the case of a simple imperative programming language without exceptions and side-effects, the following inference rules can be used to axiomatize Hoare predicate

Inference Rule I

$$\frac{\{P\}c_1\{R\} \qquad \{R\}c_2\{Q\}}{\{P\}c_1;c_2\{Q\}}$$

The above rule is valid because during execution of the program $c_1;c_2$ there is a moment when c_1 is already completely executed and execution of c_2 has not started yet.

Inference Rule II

$$\frac{\forall en : Env(P(en) \Rightarrow Q(en[v \mapsto e]))}{\{P\}v := e\{Q\}}$$

where

$$en[v \mapsto e] = \lambda a.\begin{cases} e & \text{if } a = v \\ en\ a & \text{otherwise} \end{cases}$$

The justification of this rule is based on the fact that $en[v \mapsto e]$ is the environment after the evaluation of the assignment $v := e$ if en was the environment before this evaluation.

Inference Rule III

$$\frac{\{P \wedge b\}c_1\{Q\} \qquad \{P \wedge \neg b\}c_2\{Q\}}{\{P\}\mathit{if}\ b\ \mathit{then}\ c_1\ \mathit{else}\ c_2\{Q\}}$$

This rule can be proven by splitting evaluation of the statement

$$\text{if } b \text{ then } c_1 \text{ else } c_2$$

into two cases: when b is true and when b is false before the evaluation.

[6]Hoare's original notation was $P\{c\}Q$.

Inference Rule IV

$$\frac{P \Rightarrow R \qquad \{R \wedge b\}c\{R\} \qquad R \wedge \neg b \Rightarrow Q}{\{P\}\mathit{while}\ b\ \mathit{do}\ c\{Q\}}$$

The proof of this rule can be given by induction on the number of loop cycles in the evaluation.

Using these rules, statements of the form $\{P\}c\{Q\}$ can be proven by induction on program c structure. I have used this methodology in my work on Simple Imperative Programming Language[7].

With some modifications, the same methodology can be applied to J program verification. These modifications are required because J has exceptions and side effects. Exceptions can lead to an abrupt termination of a program. In particular, if code c_1 throws an exception, the execution of the code $c_1; c_2$ will not include code c_2 execution. As a result, Hoare inference rule I, generally speaking, does not hold. To avoid this situation, I have decided to change the meaning of the Hoare triple $\{P\}c\{Q\}$ to: *if pre-condition P is true before program c is executed and* **program c terminates normally**, *then condition Q is true after the execution.* This modification makes rule I valid for J language.

Rules II, III, and IV, in general, are not valid for the J language since they do not take into account that evaluation of expressions e and b can have side-effect. Although it is possible to modify these rules to handle side-effects, I do not attempt it because the simplest programs do not use expressions with side effects. I have found that the simplest solution is to prove separate instances of Hoare rules for every expression that is used in the program and to write a tactic that will consecutively try to apply such rules until the correct one is found. Theory `hoare` contains such Hoare rules in the form of Nuprl theorems and theory `verify_1` gives examples of their application to program verification.

[7] `http://www.cs.cornell.edu /Info/Projects/NuPrl/Nuprl4.2/Libraries/Semantics`

Appendix A

Deformalization

Any work on formalization makes sense only if the results of this work are accessible to a human being. Unfortunately, formal mathematical theories usually are huge and hard to read and to understand. Computer printouts of my Java semantics theory, that can be found in the appendix to this dissertation consist of about 120 pages that are almost impossible to read without external assistance such as the previous chapter of my dissertation.

In my work I was constantly concerned about presenting formal theories in a more easy-to-read and easy-to-understand form. Two approaches to this problem that I have experimented with are proof transformation and proof publishing.

A.0.1 Proof Transformation

Proof transformation is a process of converting a formal proof into another formal proof. The new proof is equivalent to the original one from the computer point of view because it proves exactly the same result, but it has different structure - the one that is easy to read for a human being. I have developed a proof transformation program that automatically converts proofs into more readable form by finding chains of *routine* deduction steps and compressing them into one step. The decision to treat some deduction step as routines was based on the tactics that the step involves.

A.0.2 Proof Publishing

Proof publishing is a process of converting a formal proof into an informal sketch of the proof that gives the reader the understanding of how formal proof has been done. Traditionally, Nuprl theories were published in the form of LaTeX documents. The appendix provides an example of such publishing. Unfortunately, since Nuprl proofs have tree structure, publishing them in LaTeX includes converting into some linear form that can fit into traditional page structure of LaTeX documents.

I have developed a new way of publishing Nuprl proofs by converting them into sets of hyper-linked HTML pages. Since HTML pages can have an arbitrary link structure, they easily can be used to represent the tree structure of the Nuprl proofs.

Web Publishing

Nuprl collection of theorems, definitions, and other auxiliary objects about some specific topic, known as *formal theory*, is presented by my converter as a HTML table (see Figure A.1). Each row in such table represents a separate object in formal Nuprl library. Each of the theorem statements is linked to the top node of its proof tree (see Figure A.2). HTML presentation of a node in the proof tree consists of the list of hypotheses, the conclusion, the tactic that was applied on this step, and the list of subgoals to which original goal was reduced on this step. Each of these subgoals is hyper-linked to the HTML page, representing the next step in the proof.

THM	int upper properties	∀i:ℤ. ∀j:{i...}. i ≤ j
ABS	int_lower	{...i} == {j:ℤ| j ≤ i}
ML	int lower ml inc	add_set_inclusion_info 'int_lower' 'int' AbSetDForInc Auto ;;
THM	int lower wf	∀i:ℤ. {...i} ∈ 𝕌₁
THM	int lower properties	∀i:ℤ. ∀j:{...i}. j ≤ i
ABS	int_seg	{i..j⁻} == {k:ℤ| i ≤ k < j}
ML	int seg ml inc	add_set_inclusion_info 'int_seg' 'int' AbSetDForInc Auto ;;
THM	int seg wf	∀m,n:ℤ. {m..n⁻} ∈ 𝕌₁
THM	int seg properties	∀i,j:ℤ. ∀y:{i..j⁻}. i ≤ y < j

Figure A.1: Front page to the Web presentation of a formal theory

Level: Lib Thy Top:

Hypotheses:

None

Conclusion:

⊢ ∀i,j:ℤ. ∀x,y:{i..j⁻}. Dec(x = y)

Applied Tactic: (Unfold 'decidable' 0 THEN RepD ...a)

Generated subgoals:

1. x = y ∨ ¬(x = y)

Figure A.2: Proof step in the Web presentation of a formal theory

In addition, each HTML page, corresponding to a node in the proof tree, provides links to all upper nodes in the proof, as well as to the theory page and to the library page - a collection of links to all existing theories.

Possible Extensions

Although already now Web publishing provides an easy access to Nuprl formal library that is used by many people at Cornell and potentially can be used by practically anybody in the world, there are several enhancements to the converter that would make it even more convenient.

- adding search capability to the HTML library
- using loadable fonts, once they become more available, to represent special Nuprl characters. Currently such characters are displayed by using small bitmaps.
- using Java applets to display formulas. Such applets can contain extra information about formula structure and abstractions that it is using.

Appendix B

Nuprl Formal Theory Printouts

```
*C parec_begin **** Parameterized Recursive Type ****
   Theory  parec adds parametrized recursive type rules
   to Nuprl Type Theory and provides basic tactic support
   for them.
*A mono         Mono{i}(b,x.B[b; x] on T) ==
                  ∀b',b'':T → 𝕌.
                    (∀x:T. b' x ⊆ b'' x)
                    ⇒ (∀x:T. B[b'; x] ⊆ B[b''; x])
*T mono_wf

⊢ ∀T:𝕌. ∀B:(T → 𝕌) → T → 𝕌.  Mono{i}(b,x.B[b;x] on T) ∈ 𝕌'
|
BY Unfold `mono` 0 THEN Auto

*A parec        parec(A,x.B[A; x] @ a) ==  !null_abstraction{}
*D parec_ind_df
                parec_ind(<r:r:*>;<h:var>,<z:var>.<t:t:*>)
                == parec_ind{}(<r>; <h>,<z>.<t>)
*A parec_ind    parec_ind(r;h,z.t[h; z]) ==
                  !null_abstraction{}
*R parecEquality
                H,
                   ⊢ parec(b1,x1.B1 @ t1)
                  = parec(b2,x2.B2 @ t2)

                  BY parecEquality x T b

                  H ⊢ t1 = t2
                  H, x:T, b:T → 𝕌
                     ⊢ B1[b,x/b1,x1] = B2[b,x/b2,x2]
                  H ⊢ Mono{i}(b1,x1.B1 on T)
*T parec_wf

⊢ ∀T:𝕌. ∀B:(T → 𝕌) → T → 𝕌. ∀t:T.
|    Mono{i}(b,x.B[b;x] on T) ⇒ parec(b,x.B[b;x] @ t) ∈ 𝕌
|
BY Unfolds ``so_apply member`` 0 THEN
   UnivCD THENM Refine `parecEquality`
   [mk_var_arg `x'; mk_term_arg ⌈T⌉; mk_var_arg `b'] THEN Auto

*R parecMemberEquality
                H,   ⊢ b1 = b2

                  BY parecMemberEquality level{i} z

                  H ⊢ b1 = b2
                  H ⊢ parec(b,x.B @ t) = parec(b,x.B @ t)


*R parecUnroll H, r:parec(b,x.B @ t), J,   ⊢ G ext g[r/r']

                  BY parecUnroll #$i r' z u

                  H, r:parec(b,x.B @ t), J
                    r':B[(λz.parec(b,x.B @ z)),t/b,x], u:r = r'
                     ⊢ G[r'/r] ext g
```

```
*R parecMemberFormation
              H,   ⊢ parec(b,x.B @ a) ext t

                BY parecMemberFormation level{i} z

                H ⊢ B[(λz.parec(b,x.B @ z)),a/b,x] ext t
                H ⊢ parec(b,x.B @ a) ∈ ℙ
*R parecElimination
              H, t:T, J, r:parec(b,x.B @ t), J1,
                 ⊢ G ext parec_ind(t;w,z.g[(λt,r.Void)/u])

                BY parecElimination level{i} #$j1 #$j2 u w v
                   s z

                H, t:T, J, r:parec(b,x.B @ t), J1,
                   ⊢ parec(b,x.B @ t) = parec(b,x.B @ t)
                H, t:T, J, r:parec(b,x.B @ t), J1, u:t:T
                  → parec(b,x.B @ t)
                  → 𝕌, w:t:T
                  → r:{v:parec(b,x.B @ t)| u t v}
                  → G, s:T, z:B[(λs.{v:parec(b,x.B @ s)|
                                   u s v} ),s/b,x]
                   ⊢ G[s,z/t,r] ext g
*R parec_indEquality
              H,
                 ⊢ parec_ind(r1;h1,z1.t1)
                = parec_ind(r2;h2,z2.t2)

                BY parec_indEquality level{i} (λx.S[x/r1])
                   (a:A × parec(A,x.B @ a)) Z u h z

                H ⊢ r1 = r2
                H, Z:A → 𝕌, u:z:(a:A × Z a) → (z = z)
                  h:z:(a:A × Z a) → S[z/x], z:a:A
                  × B[Z/A][a/x]
                   ⊢ t1[h/h1][z/z1] = t2[h/h2][z/z2]
*C parec_end   *********************************
```

```
*C signature_begin **** SIGNATURE ****
*C This theory provides formalism for specification
   of an arbitrary reference type.
*A signature   Sign ==
                 nam:𝕌
                 × cor:(nam → 𝕌)
                 × c:(n:nam → cor n)
                 × idx:(nam → 𝕌)
                 × fld:(n:nam → idx n → nam)
                 × eq_nam:(nam → nam → 𝔹)
                 × eq_cor:(n1:nam
                        → cor n1
                        → n2:nam
                        → cor n2
                        → 𝔹)
                 × (n1:nam → idx n1 → n2:nam → idx n2 → 𝔹)
*T signature_wf

⊢ Sign ∈ 𝕌'
|
BY Unfold `signature` 0 THEN Auto

*A sign_nam                   Nam ==  s.1
*T sign_nam_wf

⊢ ∀s:Sign. Nam ∈ 𝕌
|
BY ModulePiTac 8 ``sign_nam sign_cor sign_c sign_idx sign_fld
   sign_eq_nam sign_eq_c
   or sign_eq_idx``

*A sign_cor                   Cor ==  s.2.1
*T sign_cor_wf

⊢ ∀s:Sign. Cor ∈ Nam → 𝕌
|
BY ModulePiTac 8 ``sign_nam sign_cor sign_c sign_idx sign_fld
   sign_eq_nam sign_eq_c
   or sign_eq_idx``

*A sign_c                     cor ==  s.2.2.1
*T sign_c_wf

⊢ ∀s:Sign. cor ∈ n:Nam → Cor n
|
BY ModulePiTac 8 ``sign_nam sign_cor sign_c sign_idx sign_fld
   sign_eq_nam sign_eq_c
   or sign_eq_idx``

*A sign_idx                   Idx ==  s.2.2.2.1
*T sign_idx_wf

⊢ ∀s:Sign. Idx ∈ Nam → 𝕌
|
BY ModulePiTac 8 ``sign_nam sign_cor sign_c sign_idx sign_fld
   sign_eq_nam sign_eq_c
   or sign_eq_idx``
```

```
*A sign_fld                  Fld ==  s.2.2.2.2.1
*T sign_fld_wf

⊢ ∀s:Sign. Fld ∈ n:Nam → Idx n → Nam
|
BY ModulePiTac 8 ‘‘sign_nam sign_cor sign_c sign_idx sign_fld
   sign_eq_nam sign_eq_c
   or sign_eq_idx‘‘

*A sign_eq_nam               EqNam ==  s.2.2.2.2.2.1
*T sign_eq_nam_wf

⊢ ∀s:Sign. EqNam ∈ Nam → Nam → 𝔹
|
BY ModulePiTac 8 ‘‘sign_nam sign_cor sign_c sign_idx sign_fld
   sign_eq_nam sign_eq_c
   or sign_eq_idx‘‘

*A sign_eq_cor               EqCor ==  s.2.2.2.2.2.2.1
*T sign_eq_cor_wf

⊢ ∀s:Sign. EqCor ∈ n1:Nam → Cor n1 → n2:Nam → Cor n2 → 𝔹
|
BY ModulePiTac 8 ‘‘sign_nam sign_cor sign_c sign_idx sign_fld
   sign_eq_nam sign_eq_c
   or sign_eq_idx‘‘

*A sign_eq_idx               EqIdx ==  s.2.2.2.2.2.2.2
*T sign_eq_idx_wf

⊢ ∀s:Sign. EqIdx ∈ n1:Nam → Idx n1 → n2:Nam → Idx n2 → 𝔹
|
BY ModulePiTac 8 ‘‘sign_nam sign_cor sign_c sign_idx sign_fld
   sign_eq_nam sign_eq_c
   or sign_eq_idx‘‘

*M create_signature
               Class Declaration for: s ∈ Sign

                 Long Name: signature
                 Short Name: sign

                 Parameters:

                 Fields:
                    (Nam)  nam : 𝕌
                    (Cor)  cor : nam → 𝕌
                    (cor)  c : n:nam → cor n
                    (Idx)  idx : nam → 𝕌
                    (Fld)  fld : n:nam → idx n → nam
                    (EqNam)  eq_nam : nam → nam → 𝔹
                    (EqCor)  eq_cor : n1:nam
                                  → cor n1
                                  → n2:nam
                                  → cor n2
                                  → 𝔹
```

```
                    (EqIdx)  eq_idx : n1:nam
                                      → idx n1
                                      → n2:nam
                                      → idx n2
                                      → 𝔹

                 Universe: 𝕌'
*A eq_nam                    (n1 =c n2) ==  EqNam n1 n2
*T eq_nam_wf

⊢ ∀s:Sign. ∀n,k:Nam.  (n =c k) ∈ 𝔹
|
BY Unfold ‘eq_nam‘ 0 THEN Auto

*A eq_cor                    (c1 =c c2) ==  EqCor n1 c1 n2 c2
*T eq_cor_wf

⊢ ∀s:Sign. ∀n1,n2:Nam. ∀c1:Cor n1. ∀c2:Cor n2.  (c1 =c c2) ∈ 𝔹
|
BY Unfold ‘eq_cor‘ 0 THEN Auto

*A eq_idx                    (i1 =c i2) ==  EqIdx n1 i1 n2 i2
*T eq_idx_wf

⊢ ∀s:Sign. ∀n1,n2:Nam. ∀i1:Idx n1. ∀i2:Idx n2.  (i1 =c i2) ∈ 𝔹
|
BY Unfold ‘eq_idx‘ 0 THEN Auto

*A reg_sign    Reg(s) ==
                 (∀n,k:Nam.  ↑(n =c k) ⟺ n = k)
                 ∧ (∀n,k:Nam. ∀c:Cor n. ∀d:Cor k.
                      ↑(c =c d) ⟺ (n = k) c∧ (c = d))
                 ∧ (∀n,k:Nam. ∀i:Idx n. ∀j:Idx k.
                      ↑(i =c j) ⟺ (n = k) c∧ (i = j))
*T reg_sign_wf

⊢ ∀s:Sign. Reg(s) ∈ ℙ
|
BY Unfold ‘reg_sign‘ 0 THEN Auto
|\
| 1. s: Sign
| 2. n: Nam
| 3. k: Nam
| 4. c: Cor n
| 5. d: Cor k
| 6. n = k
| ⊢ d ∈ Cor n
| |
1 BY SubstClause ⌈Cor n⌉ 5 THENA EqCD THEN Auto
 \
  1. s: Sign
  2. n: Nam
  3. k: Nam
  4. i: Idx n
  5. j: Idx k
```

```
  6. n = k
  ⊢ j ∈ Idx n
  |
  BY SubstClause ⌈Idx n⌉ 5 THENA EqCD THEN Auto

*C signature_end
              **********************************
```

```
*C class_begin   ************ Class ************
*C For any signature this theory defines a parametrized
   recursive type in standard Nuprl Type Theory that
   models reference type, corresponding to the signature.
*A class       ρ@n ==
                 parec(C,j.{k:Nam| k = j}  + k:Nam
                 × C k
                 × Cor k
                 × C j + k:Nam
                 × C k
                 × i:Idx k
                 × C (Fld k i)
                 × C j @ n)
*T class_wf

⊢ ∀s:Sign. ∀n:Nam.  ρ@n ∈ 𝕌
|
BY Unfold ‘class‘ 0 THEN Auto
|
1. s: Sign
2. n: Nam
⊢ Mono{i}(C,j.{k:Nam| k = j}  + k:Nam
|  × C k
|  × Cor k
|  × C j + k:Nam × C k × i:Idx k × C (Fld k i) × C j on Nam)
|
BY Unfold ‘mono‘ 0 THEN Auto THEN BHyp 5 THEN Auto

*A nil_object                ⊕(k) ==  inl k
*T nil_object_wf

⊢ ∀s:Sign. ∀n,k:Nam.  k = n ⇒ ⊕(k) ∈ ρ@n
|
BY Unfold ‘nil_object‘ 0 THEN UnivCD  THENM AbParecMemTypeCD T
   HEN Auto

*A cor_update  [core(x:ρ@k):=t@y] ==  inr (inl <k, x, t, y> )
*T cor_update_wf

⊢ ∀s:Sign. ∀n,k:Nam. ∀x:ρ@k. ∀t:Cor k. ∀y:ρ@n.
|    [core(x:ρ@k):=t@y] ∈ ρ@n
|
BY Unfold ‘cor_update‘ 0 THEN UnivCD THENM AbParecMemTypeCD TH
   EN Auto

*A ref_update  [(x:ρ@k).i:=z@y] ==  inr inr <k, x, i, z, y>
*T ref_update_wf

⊢ ∀s:Sign. ∀n,k:Nam. ∀x:ρ@k. ∀i:Idx k. ∀z:ρ@Fld k i. ∀y:ρ@n.
|    [(x:ρ@k).i:=z@y] ∈ ρ@n
|
BY Unfold ‘ref_update‘ 0 THEN UnivCD THENM AbParecMemTypeCD TH
   EN Auto

*A object_cases
             case o
```

```
                of ⊕(n) -> base_case[n]

                  | [core(x:ρ@k):=t@y] -> cor_case[k; x; t; y]

                  | [(u:ρ@m).i:=v@w] -> ref_case[m; u; i; v; w
                ]

                ) ==
                  case o
                  of inl(n) => base_case[n]
                   | inr(cr) => case cr
                                of inl(c) => let <k,xty> = c
                                             in
                                             let <x,ty> = xty
                                             in
                                             let <t,y> = ty
                                             in
                                             cor_case[k; x; t;
                y]
                                 | inr(r) => let <m,uivw> = r
                                             in
                                             let <u,ivw> = uivw
                                             in
                                             let <i,vw> = ivw
                                             in
                                             let <v,w> = vw
                                             in
                                             ref_case[m; u; i;
                v; w]
*T object_cases_wf

⊢ ∀T:𝕌. ∀s:Sign. ∀n:Nam. ∀o:ρ@n. ∀base_case:k:Nam → T.
|  ∀cor_case:k:Nam → ρ@k → Cor k → ρ@n → T.
|  ∀ref_case:k:Nam → ρ@k → i:Idx k → ρ@Fld k i → ρ@n → T.
|    case o
|
|   of ⊕(k) -> base_case[k]
|
|    | [core(x:ρ@k):=t@y] -> cor_case[k;x;t;y]
|
|    | [(x:ρ@k).i:=z@y] -> ref_case[k;x;i;z;y]
|
|  ) ∈ T
|
BY Unfold `object_cases` 0 THEN UnivCD THENM AbParecTypeHD 4 T
   HEN Auto

*T class_ind_tp

⊢ ∀s:Sign. ∀P:n:Nam → ρ@n → ℙ.
|    (∀n,k:Nam.  k = n ⇒ P[n;⊕(k)])
|    ⇒ (∀n,k:Nam. ∀x:ρ@k. ∀t:Cor k. ∀y:ρ@n.
|          P[k;x] ⇒ P[n;y] ⇒ P[n;[core(x:ρ@k):=t@y]])
|    ⇒ (∀n,k:Nam. ∀x:ρ@k. ∀i:Idx k. ∀z:ρ@Fld k i. ∀y:ρ@n.
|          P[k;x]
```

```
|           ⇒ P[Fld k i;z]
|           ⇒ P[n;y]
|           ⇒ P[n;[(x:ρ@k).i:=z@y]])
|     ⇒ {∀n:Nam. ∀o:ρ@n.  P[n;o]}
|
BY %E% Unfolds ‘‘so_apply guard‘‘ 0 THEN UnivCD THENA Auto
|
1. s: Sign
2. P: n:Nam → ρ@n → ℙ
3. ∀n,k:Nam.  k = n ⇒ P n ⊕(k)
4. ∀n,k:Nam. ∀x:ρ@k. ∀t:Cor k. ∀y:ρ@n.
     P k x ⇒ P n y ⇒ P n [core(x:ρ@k):=t@y]
5. ∀n,k:Nam. ∀x:ρ@k. ∀i:Idx k. ∀z:ρ@Fld k i. ∀y:ρ@n.
     P k x ⇒ P (Fld k i) z ⇒ P n y ⇒ P n [(x:ρ@k).i:=z@y]
6. n: Nam
7. o: ρ@n
⊢ P n o
|
BY PushArgs [‘hr_n‘, int_to_arg 6;
|  ‘hr_o‘, int_to_arg 7]
|
|
BY %S% \p.AbParecInd (get_int_arg ‘hr_n‘ p) (get_int_arg ‘hr_o
|  ‘ p) p
|\
| ⊢ ρ@n ∈ 𝕌
| |
1 BY AddHiddenLabel ‘wf‘
| |
| |
1 BY %E% Auto
 \
  8. Q: n:Nam → ρ@n → ℙ
  9. ∀n:Nam. ∀o:{v:ρ@n| Q n v} .  P n o
  10. n1: Nam
  11. o1: {k:Nam| k = n1}  + k:Nam
          × (λn1.{v:ρ@n1| Q n1 v} ) k
          × Cor k
          × (λn1.{v:ρ@n1| Q n1 v} ) n1 + k:Nam
          × (λn1.{v:ρ@n1| Q n1 v} ) k
          × i:Idx k
          × (λn1.{v:ρ@n1| Q n1 v} ) (Fld k i)
          × (λn1.{v:ρ@n1| Q n1 v} ) n1
  ⊢ P n1 o1
  |
  BY D (-1)
  |\
  | 11. x: {k:Nam| k = n1}
  | ⊢ P n1 (inl x )
  | |
  1 BY Fold ‘nil_object‘ 0 THEN OnHyps [-3;-3] Thin THEN Renam
  | |  eVarUsingArg ‘v1‘ ‘n’ (-1
  | |  )
```

```
| |
| 8. n1: Nam
| 9. n2: {k:Nam| k = n1}
| ⊢ P n1 ⊕(n2)
| |
1 BY %E% D (9) THEN BHyp 3 THEN Auto
 \
  11. y: k:Nam
         × (λn1.{v:ρ@n1| Q n1 v} ) k
         × Cor k
         × (λn1.{v:ρ@n1| Q n1 v} ) n1 + k:Nam
         × (λn1.{v:ρ@n1| Q n1 v} ) k
         × i:Idx k
         × (λn1.{v:ρ@n1| Q n1 v} ) (Fld k i)
         × (λn1.{v:ρ@n1| Q n1 v} ) n1
  ⊢ P n1 (inr y )
  |
  BY Reduce (-1) THEN D (-1) THEN D (-1) THEN D (-1) THEN D
  |  (-1)
  |\
  | 11. k: Nam
  | 12. x2: {v:ρ@k| Q k v}
  | 13. x4: Cor k
  | 14. x5: {v:ρ@n1| Q n1 v}
  | ⊢ P n1 (inr (inl <k, x2, x4, x5> ) )
  | |
  1 BY \p.(Fold ‘cor_update‘ 0 THEN InstHyp [mvt (var_of_hyp
  | |   (-4) p);mvt (var_of_hyp (-
  | |  3) p)] (-6)) p
  | |\
  | | ⊢ k ∈ Nam
  | | |
  1 2 BY NthDecl (-4)
  | |\
  | | ⊢ x2 ∈ {v:ρ@k| Q k v}
  | | |
  1 2 BY NthDecl (-3)
  |  \
  |   15. P k x2
  |   ⊢ P n1 [core(x2:ρ@k):=x4@x5]
  |   |

  1   BY \p. (InstHyp [mvt (var_of_hyp (-6) p); mvt (var_of_
  |   |  hyp (-2) p)] (-7)) p
  |   |\
  |   | ⊢ n1 ∈ Nam
  |   | |
  1   2 BY NthDecl (-6)
  |   |\
  |   | ⊢ x5 ∈ {v:ρ@n1| Q n1 v}
  |   | |
  1   2 BY NthDecl (-2)
  |    \
  |     16. P n1 x5
```

```
|     |
1     BY D (-3) THEN Thin (-3) THEN D (-5) THEN Thin (-5)
|     |  THEN Thin (-8) THEN Thin (-8)
|     |
|     8. n1: Nam
|     9. k: Nam
|     10. x2: ρ@k
|     11. x4: Cor k
|     12. x5: ρ@n1
|     13. P k x2
|     14. P n1 x5
|     |
1     BY %E% BHyp 4 THEN Auto
 \
  11. k: Nam
  12. y3: {v:ρ@k| Q k v}
  13. i: Idx k
  14. y5: {v:ρ@Fld k i| Q (Fld k i) v}
          × {v:ρ@n1| Q n1 v}
  ⊢ P n1 (inr inr <k, y3, i, y5>  )
  |
  BY D (-1)
  |
  14. y6: {v:ρ@Fld k i| Q (Fld k i) v}
  15. y7: {v:ρ@n1| Q n1 v}
  ⊢ P n1 (inr inr <k, y3, i, y6, y7>  )
  |
  BY \p.(Fold `ref_update` 0 THEN InstHyp [mvt (var_of_hyp
  |   (-5) p);mvt (var_of_hyp (-
  |  4) p)] (-7)) p
  |\
  | ⊢ k ∈ Nam
  | |
  1 BY NthDecl (-5)
  |\
  | ⊢ y3 ∈ {v:ρ@k| Q k v}
  | |
  1 BY NthDecl (-4)
   \
    16. P k y3
    ⊢ P n1 [(y3:ρ@k).i:=y6@y7]
    |
    BY \p.(InstHyp [mvt (var_of_hyp (-7) p);mvt (var_of_hy
    |  p (-2) p)] (-8)) p
    |\
    | ⊢ n1 ∈ Nam
    | |
    1 BY NthDecl (-7)
    |\
    | ⊢ y7 ∈ {v:ρ@n1| Q n1 v}
    | |
    1 BY NthDecl (-2)
     \
```

```
17. P n1 y7
|
BY D (-4)
|
14. y6: ρ@Fld k i
[15]. Q (Fld k i) y6
16. y7: {v:ρ@n1| Q n1 v}
17. P k y3
18. P n1 y7
|
BY \p. (let i = get_pos_hyp_num (-5) p in
|  let v, T = dest_hyp i p in
|  InstHyp [snd(hd(tl(snd(dest_term T)))); mvt (var_
|  of_hyp (-5) p)] (-10))
|  p
|\
| 15. Q (Fld k i) y6
| ⊢ Fld k i ∈ Nam
| |
1 BY RepeatFor 5 (Thin (-1)) THEN Thin (-2) THEN Rep
| |  eatFor 3 (Thin (-3))
| |
| 8. k: Nam
| 9. i: Idx k
| |
1 BY MemCD
| |\
| | ⊢ Fld k ∈ Idx k → Nam
| | |
1 2 BY MemCD
| | |\
| | | ⊢ Fld ∈ n:Nam → Idx n → Nam
| | | |
1 2 3 BY MemCD THEN AddHiddenLabel ‘wf‘
| | | |

| | | ⊢ s ∈ Sign
| | | |
1 2 3 BY %E% Auto
| |  \
| |   ⊢ k ∈ Nam
| |   |
1 2   BY NthDecl (-2)
|  \
|   ⊢ i ∈ Idx k
|   |
1   BY NthDecl (-1)
|\
| 15. Q (Fld k i) y6
| ⊢ y6 ∈ {v:ρ@Fld k i| Q (Fld k i) v}
| |
1 BY MemTypeCD
| |\
| | ⊢ y6 ∈ ρ@Fld k i
```

```
| | |
1 2 BY NthDecl (-5)
| |\
| | ⊢ Q (Fld k i) y6
| | |
1 2 BY Trivial
|  \
|   19. v: ρ@Fld k i
|   ⊢ Q (Fld k i) v ∈ 𝕌
|   |
1   BY MemCD
|   |\
|   | ⊢ Q (Fld k i) ∈ ρ@Fld k i → ℙ
|   | |
1   2 BY MemCD
|   | |\
|   | | ⊢ Q ∈ n:Nam → ρ@n → ℙ
|   | | |
1   2 3 BY NthDecl (-12)
|   |  \
|   |   ⊢ Fld k i ∈ Nam
|   |   |
1   2   BY MemCD
|   |   |\
|   |   | ⊢ Fld k ∈ Idx k → Nam
|   |   | |
1   2   3 BY MemCD
|   |   | |\
|   |   | | ⊢ Fld ∈ n:Nam → Idx n → Nam
|   |   | | |
1   2   3 4 BY MemCD THEN AddHiddenLabel ‘wf‘
|   |   | | |
|   |   | | ⊢ s ∈ Sign
|   |   | | |
1   2   3 4 BY %E% Auto
|   |   |  \
|   |   |   ⊢ k ∈ Nam
|   |   |   |
1   2   3   BY NthDecl (-9)
|   |    \
|   |     ⊢ i ∈ Idx k
|   |     |
1   2     BY NthDecl (-7)
|   |\
|   | ⊢ v ∈ ρ@Fld k i
|   | |
1   2 BY NthDecl (-1)
|    \
|     20. Q (Fld k i) v = Q (Fld k i) v
|     ⊢ ℙ ∈ 𝕌’
|     |
1     BY MemCD
 \
```

```
            19. P (Fld k i) y6
            |
            BY D (-4) THEN Thin (-4) THEN Thin (-5) THEN D (-7
            |  ) THEN Thin (-7) THEN Thin (-10)
            |  THEN Thin (-10)
            |
            8. n1: Nam
            9. k: Nam
            10. y3: ρ@k
            11. i: Idx k
            12. y6: ρ@Fld k i
            13. y7: ρ@n1
            14. P k y3
            15. P n1 y7
            16. P (Fld k i) y6
            |
            BY %E% BHyp 5 THEN Auto

*M nam_ml      nam(o)
               ==r case o

                of ⊕(k) -> k

                 | [core(x:ρ@k):=t@y] -> nam(y)

                 | [(x:ρ@k).i:=z@y] -> nam(y)

               )
*T nam_wf

⊢ ∀s:Sign. ∀n:Nam. ∀o:ρ@n.  nam(o) ∈ Nam
|
BY UnivCD THENM ClassInd 2 3 THENM RecCaseSplit ‘nam‘ THEN Aut
   o

*T nam_sound

⊢ ∀s:Sign. ∀n:Nam. ∀o:ρ@n.  nam(o) = n
|
BY UnivCD THENM ClassInd 2 3 THENM Reduce 0 THENML [D (-1); Id
   ; Id] THEN Auto

*T nam_sound_mem

⊢ ∀s:Sign. ∀n:Nam. ∀x:ρ@n.  x ∈ ρ@nam(x)
|
BY UnivCD THENA Auto
|
1. s: Sign
2. n: Nam
3. x: ρ@n
⊢ x ∈ ρ@nam(x)
|
BY InstLemma ‘par_type_lemma‘ [⌈Nam⌉;⌈λ₂m.ρ@m⌉;⌈n⌉;⌈nam(x)⌉;⌈x
|  ⌉] THEN Auto
|
```

```
⊢ n = nam(x)
|
BY RWH (LemmaC ‘nam_sound‘) 0 THEN Auto

*M eq_obj_ml
              (o1=₀o2)
              ==r case o1

               of ⊕(k1) -> case o2

               of ⊕(k2) -> (k1 =c k2)

                | [core(x2:ρ@k2):=t2@y2] -
              > ff

                | [(x2:ρ@k2).i2:=z2@y2] ->
               ff

              )

                | [core(x1:ρ@k1):=t1@y1] -> case o2

               of ⊕(k2) -
              > ff

                | [core(x
              2:ρ@k2):=t2@y2] -> (k1 =c k2)
                                                 ∧b (x1=₀x2)
                                                 ∧b (t1 =c t2
              )
                                                 ∧b (y1=₀y2)

                | [(x2:ρ@
              k2).i2:=z2@y2] -> ff

              )

                | [(x1:ρ@k1).i1:=z1@y1] -> case o2

               of ⊕(k2) ->
               ff

                | [core(x2
              :ρ@k2):=t2@y2] -> ff

                | [(x2:ρ@k
              2).i2:=z2@y2] -> (k1 =c k2)
                                                ∧b (x1=₀x2)
                                                ∧b (i1 =c i2)
                                                ∧b (z1=₀z2)
                                                ∧b (y1=₀y2)

              )

              )
*T eq_obj_wf
```

```
⊢ ∀s:Sign. ∀n:Nam. ∀x:ρ@n. ∀k:Nam. ∀y:ρ@k.  (x=₀y) ∈ 𝔹
|
BY RepeatMFor 3 (D 0) THENM ClassInd 2 3 THENM UnivCD THENM Cl
|  assInd (-2) (-1) THEN
|  M RecCaseSplit ‘eq_obj‘ THEN Auto
|\
| 1. s: Sign
| 2. n: Nam
| 3. x: ρ@n
| 4. n1: Nam
| 5. k: Nam
| 6. x3: ρ@k
| 7. x5: Cor k
| 8. x6: ρ@n1
| 9. ∀k:Nam. ∀y:ρ@k.  (x3=₀y) ∈ 𝔹
| 10. ∀k:Nam. ∀y:ρ@k.  (x6=₀y) ∈ 𝔹
| 11. k@0: Nam
| 12. y@0: ρ@k@0
| 13. k1: Nam
| 14. k2: Nam
| 15. x9: ρ@k2
| 16. x11: Cor k2
| 17. x12: ρ@k1
| 18. ([core(x3:ρ@k):=x5@x6]=₀x9) ∈ 𝔹
| 19. ([core(x3:ρ@k):=x5@x6]=₀x12) ∈ 𝔹
| ⊢ (x3=₀x9) ∈ 𝔹
| |
1 BY BHyp 9 THEN Auto
|\
| 1. s: Sign
| 2. n: Nam
| 3. x: ρ@n
| 4. n1: Nam
| 5. k: Nam
| 6. x3: ρ@k
| 7. x5: Cor k
| 8. x6: ρ@n1
| 9. ∀k:Nam. ∀y:ρ@k.  (x3=₀y) ∈ 𝔹
| 10. ∀k:Nam. ∀y:ρ@k.  (x6=₀y) ∈ 𝔹
| 11. k@0: Nam
| 12. y@0: ρ@k@0
| 13. k1: Nam
| 14. k2: Nam
| 15. x9: ρ@k2
| 16. x11: Cor k2
| 17. x12: ρ@k1
| 18. ([core(x3:ρ@k):=x5@x6]=₀x9) ∈ 𝔹
| 19. ([core(x3:ρ@k):=x5@x6]=₀x12) ∈ 𝔹
| ⊢ (x6=₀x12) ∈ 𝔹
| |
1 BY BHyp 10 THEN Auto
|\
| 1. s: Sign
```

```
| 2. n: Nam
| 3. x: ρ@n
| 4. n1: Nam
| 5. k: Nam
| 6. y3: ρ@k
| 7. i: Idx k
| 8. y6: ρ@Fld k i
| 9. y7: ρ@n1
| 10. ∀k:Nam. ∀y:ρ@k.  (y3=₀y) ∈ 𝔹
| 11. ∀k:Nam. ∀y:ρ@k.  (y7=₀y) ∈ 𝔹
| 12. ∀k:Nam. ∀y:ρ@k.  (y6=₀y) ∈ 𝔹
| 13. k@0: Nam
| 14. y@0: ρ@k@0
| 15. k1: Nam
| 16. k2: Nam
| 17. y10: ρ@k2
| 18. i1: Idx k2
| 19. y13: ρ@Fld k2 i1
| 20. y14: ρ@k1
| 21. ([(y3:ρ@k).i:=y6@y7]=₀y10) ∈ 𝔹
| 22. ([(y3:ρ@k).i:=y6@y7]=₀y14) ∈ 𝔹
| 23. ([(y3:ρ@k).i:=y6@y7]=₀y13) ∈ 𝔹
| ⊢ (y3=₀y10) ∈ 𝔹
| |
1 BY BHyp 10 THEN Auto
|\
| 1. s: Sign
| 2. n: Nam
| 3. x: ρ@n
| 4. n1: Nam
| 5. k: Nam
| 6. y3: ρ@k
| 7. i: Idx k
| 8. y6: ρ@Fld k i
| 9. y7: ρ@n1
| 10. ∀k:Nam. ∀y:ρ@k.  (y3=₀y) ∈ 𝔹
| 11. ∀k:Nam. ∀y:ρ@k.  (y7=₀y) ∈ 𝔹
| 12. ∀k:Nam. ∀y:ρ@k.  (y6=₀y) ∈ 𝔹
| 13. k@0: Nam
| 14. y@0: ρ@k@0
| 15. k1: Nam
| 16. k2: Nam
| 17. y10: ρ@k2
| 18. i1: Idx k2
| 19. y13: ρ@Fld k2 i1
| 20. y14: ρ@k1
| 21. ([(y3:ρ@k).i:=y6@y7]=₀y10) ∈ 𝔹

| 22. ([(y3:ρ@k).i:=y6@y7]=₀y14) ∈ 𝔹
| 23. ([(y3:ρ@k).i:=y6@y7]=₀y13) ∈ 𝔹
| ⊢ (y6=₀y13) ∈ 𝔹
| |
1 BY BHyp 12 THEN Auto
 \
```

```
  1. s: Sign
  2. n: Nam
  3. x: ρ@n
  4. n1: Nam
  5. k: Nam
  6. y3: ρ@k
  7. i: Idx k
  8. y6: ρ@Fld k i
  9. y7: ρ@n1
  10. ∀k:Nam. ∀y:ρ@k.  (y3=0y) ∈ 𝔹
  11. ∀k:Nam. ∀y:ρ@k.  (y7=0y) ∈ 𝔹
  12. ∀k:Nam. ∀y:ρ@k.  (y6=0y) ∈ 𝔹
  13. k@0: Nam
  14. y@0: ρ@k@0
  15. k1: Nam
  16. k2: Nam
  17. y10: ρ@k2
  18. i1: Idx k2
  19. y13: ρ@Fld k2 i1
  20. y14: ρ@k1
  21. ([(y3:ρ@k).i:=y6@y7]=0y10) ∈ 𝔹
  22. ([(y3:ρ@k).i:=y6@y7]=0y14) ∈ 𝔹
  23. ([(y3:ρ@k).i:=y6@y7]=0y13) ∈ 𝔹
  ⊢ (y7=0y14) ∈ 𝔹
  |
  BY BHyp 11 THEN Auto

*T eq_obj_imp_eq_nam

⊢ ∀s:Sign. ∀n:Nam. ∀x:ρ@n. ∀k:Nam. ∀y:ρ@k.
|    Reg(s) ⇒ ↑(x=0y) ⇒ n = k
|
BY RepeatMFor 3 (D 0) THENA Auto
|
1. s: Sign
2. n: Nam
3. x: ρ@n
⊢ ∀k:Nam. ∀y:ρ@k.  Reg(s) ⇒ ↑(x=0y) ⇒ n = k
|
BY ClassInd 2 3 THENM D 0 THENM D 0 THENM ClassIndG (-2) (-1)
|  THENM Reduce 0 THEN A
|  uto
|\
| 4. n1: Nam
| 5. n2: {k:Nam| k = n1}
| 6. k: Nam
| 7. y: ρ@k
| 8. k1: Nam
| 9. n3: {k:Nam| k = k1}
| 10. Reg(s)
| 11. ↑(n2 =c n3)
| ⊢ n1 = k1
| |
1 BY D 10
```

```
| |
| 10. ∀n,k:Nam.  ↑(n =c k) ⟺ n = k
| 11. (∀n,k:Nam. ∀c:Cor n. ∀d:Cor k.
|         ↑(c =c d) ⟺ (n = k) c∧ (c = d))
|     ∧ (∀n,k:Nam. ∀i:Idx n. ∀j:Idx k.
|           ↑(i =c j) ⟺ (n = k) c∧ (i = j))
| 12. ↑(n2 =c n3)
| |
1 BY FHyp 10 [12] THENM D 5 THENM D 10 THEN Auto
|\
| 4. n1: Nam
| 5. k: Nam
| 6. x3: ρ@k
| 7. x5: Cor k
| 8. x6: ρ@n1
| 9. ∀k@0:Nam. ∀y:ρ@k@0.  Reg(s) ⇒ ↑(x3=0y) ⇒ k = k@0
| 10. ∀k:Nam. ∀y:ρ@k.  Reg(s) ⇒ ↑(x6=0y) ⇒ n1 = k
| 11. k@0: Nam
| 12. y@0: ρ@k@0
| 13. k1: Nam
| 14. k2: Nam
| 15. x9: ρ@k2
| 16. x11: Cor k2
| 17. x12: ρ@k1
| 18. Reg(s) ⇒ ↑([core(x3:ρ@k):=x5@x6]=0x9) ⇒ n1 = k2
| 19. Reg(s) ⇒ ↑([core(x3:ρ@k):=x5@x6]=0x12) ⇒ n1 = k1
| 20. Reg(s)
| 21. ↑((k =c k2) ∧b (x3=0x9) ∧b (x5 =c x11) ∧b (x6=0x12))
| ⊢ n1 = k1
| |
1 BY RWH (bool_to_propC) 21 THENM RepeatMFor 3 (D (-1)) THENA
| |  Auto
| |
| 21. ↑(k =c k2)
| 22. ↑(x3=0x9)
| 23. ↑(x5 =c x11)
| 24. ↑(x6=0x12)
| |
1 BY FHyp 10 [20;24] THEN Auto
 \
  4. n1: Nam
  5. k: Nam
  6. y3: ρ@k
  7. i: Idx k
  8. y6: ρ@Fld k i
  9. y7: ρ@n1
  10. ∀k@0:Nam. ∀y:ρ@k@0.  Reg(s) ⇒ ↑(y3=0y) ⇒ k = k@0
  11. ∀k:Nam. ∀y:ρ@k.  Reg(s) ⇒ ↑(y7=0y) ⇒ n1 = k
  12. ∀k@0:Nam. ∀y:ρ@k@0.  Reg(s) ⇒ ↑(y6=0y) ⇒ Fld k i = k@0
  13. k@0: Nam
  14. y@0: ρ@k@0
  15. k1: Nam
  16. k2: Nam
```

```
  17. y10: ρ@k2
  18. i1: Idx k2
  19. y13: ρ@Fld k2 i1
  20. y14: ρ@k1
  21. Reg(s) ⇒ ↑([(y3:ρ@k).i:=y6@y7]=_0y10) ⇒ n1 = k2
  22. Reg(s) ⇒ ↑([(y3:ρ@k).i:=y6@y7]=_0y14) ⇒ n1 = k1
  23. Reg(s) ⇒ ↑([(y3:ρ@k).i:=y6@y7]=_0y13) ⇒ n1 = Fld k2 i1
  24. Reg(s)
  25. ↑((k =_c k2)
      ∧_b (y3=_0y10)
      ∧_b (i =_c i1)
      ∧_b (y6=_0y13)
      ∧_b (y7=_0y14))
  ⊢ n1 = k1
  |
  BY RWH bool_to_propC 25 THENM RepeatMFor 4 (D (-1)) THENA Au
  |  to
  |
  25. ↑(k =_c k2)
  26. ↑(y3=_0y10)
  27. ↑(i =_c i1)
  28. ↑(y6=_0y13)

  29. ↑(y7=_0y14)
  |
  BY FHyp 11 [24;29] THEN Auto

*T par_type_lemma

⊢ ∀T:𝕌. ∀F:T → 𝕌. ∀t,t':T. ∀f:F[t].  t = t' ⇒ f ∈ F[t']
|
BY UnivCD THENA Auto
|
1. T: 𝕌
2. F: T → 𝕌
3. t: T
4. t': T
5. f: F[t]
6. t = t'
⊢ f ∈ F[t']
|
BY SubstClause ⌈F[t']⌉ 5 THENA EqCD THEN Auto

*M cor_ml       cor(o)
                ==r case o

                 of ⊕(n) -> cor n

                  | [core(x:ρ@k):=t@y] -> if (x=_0y)
                    then t
                    else cor(y)
                    fi

                  | [(x:ρ@k).i:=z@y] -> cor(y)
```

```
                )
*T cor_wf

⊢ ∀s:Sign. ∀n:Nam. ∀o:ρ@n.  Reg(s) ⇒ cor(o) ∈ Cor n
|
BY UnivCD THENM ClassInd 2 3 THENM Reduce 0 THENA Auto
|\
| 1. s: Sign
| 2. n: Nam
| 3. o: ρ@n
| 4. Reg(s)
| 5. n1: Nam
| 6. n2: {k:Nam| k = n1}
| ⊢ cor(⊕(n2)) ∈ Cor n1
| |
1 BY RecCaseSplit `cor` THEN Auto
| |
| ⊢ cor n2 ∈ Cor n1
| |
1 BY D 6 THEN InstLemma `par_type_lemma` [⌈Nam⌉;⌈λ2t.Cor t⌉; ⌈
|    n2⌉;⌈n1⌉;⌈cor n2⌉
|    ] THEN Auto
|\
| 1. s: Sign
| 2. n: Nam
| 3. o: ρ@n
| 4. Reg(s)
| 5. n1: Nam
| 6. k: Nam
| 7. x2: ρ@k
| 8. x4: Cor k
| 9. x5: ρ@n1
| 10. cor(x2) ∈ Cor k
| 11. cor(x5) ∈ Cor n1
| ⊢ cor([core(x2:ρ@k):=x4@x5]) ∈ Cor n1
| |
1 BY RecCaseSplit `cor` THEN SplitOnConclITE THENA Auto
| |\
| | 12. ↑(x2=0x5)
| | ⊢ x4 ∈ Cor n1
| | |
1 2 BY SubstClause ⌈Cor n1⌉ 8 THENA EqCD THEN Auto
| | |
| | ⊢ k = n1
| | |
1 2 BY FLemma `eq_obj_imp_eq_nam` [4;12] THEN Auto
|  \
|   12. ¬↑(x2=0x5)
|   ⊢ cor(x5) ∈ Cor n1
|   |
1   BY Auto
 \
  1. s: Sign
  2. n: Nam
```

```
  3. o: ρ@n
  4. Reg(s)
  5. n1: Nam
  6. k: Nam
  7. y3: ρ@k
  8. i: Idx k
  9. y6: ρ@Fld k i
  10. y7: ρ@n1
  11. cor(y3) ∈ Cor k
  12. cor(y7) ∈ Cor n1
  13. cor(y6) ∈ Cor (Fld k i)
  ⊢ cor([(y3:ρ@k).i:=y6@y7]) ∈ Cor n1
  |
  BY RecCaseSplit ‘cor‘ THEN Auto

*M ref_ml
             o.i
             ==r case o

              of ⊕(k) -> ⊕(Fld k i)

                | [core(x:ρ@k):=t@y] -> [core(x:ρ@k):=t@
             y.i]

                | [(x:ρ@k).j:=z@y] -> if (x=_0y)
                                            ∧_b (j =_c i)
                  then [(x:ρ@k).j:=z@z]
                  else [(x:ρ@k).j:=z@y.i]
                  fi

             )
*T ref_wf

⊢ ∀s:Sign. ∀n:Nam. ∀o:ρ@n. ∀i:Idx n.
|    Reg(s) ⇒ o.i ∈ ρ@Fld n i
|
BY RepeatMFor 3 (D 0) THENM ClassInd 2 3 THENA Auto
|\
| 1. s: Sign
| 2. n: Nam
| 3. o: ρ@n
| 4. n1: Nam
| 5. n2: {k:Nam| k = n1}
| ⊢ ∀i:Idx n1. Reg(s) ⇒ ⊕(n2).i ∈ ρ@Fld n1 i
| |
1 BY D 5 THEN D 0 THENM Assert ⌈i ∈ Idx n2⌉ THENA Auto
| |\
| | 5. n2: Nam
| | [6]. n2 = n1
| | 7. i: Idx n1
| | ⊢ i ∈ Idx n2
| | |
1 2 BY SubstClause ⌈Idx n2⌉ 7 THENA EqCD THEN Auto
|   \
|    5. n2: Nam
```

```
|   [6]. n2 = n1
|   7. i: Idx n1
|   8. i ∈ Idx n2
|   ⊢ Reg(s) ⇒ ⊕(n2).i ∈ ρ@Fld n1 i
|   |
1   BY RecCaseSplit ‘ref‘ THEN Auto
|\
| 1. s: Sign
| 2. n: Nam
| 3. o: ρ@n
| 4. n1: Nam
| 5. k: Nam
| 6. x2: ρ@k
| 7. x4: Cor k
| 8. x5: ρ@n1
| 9. ∀i:Idx k. Reg(s) ⇒ x2.i ∈ ρ@Fld k i
| 10. ∀i:Idx n1. Reg(s) ⇒ x5.i ∈ ρ@Fld n1 i
| ⊢ ∀i:Idx n1. Reg(s) ⇒ [core(x2:ρ@k):=x4@x5].i ∈ ρ@Fld n1 i
| |
1 BY RecCaseSplit ‘ref‘ THEN Auto
| |
| 11. i: Idx n1
| 12. Reg(s)
| ⊢ x5.i ∈ ρ@Fld n1 i
| |
1 BY BHyp 10 THEN Auto
 \
  1. s: Sign
  2. n: Nam
  3. o: ρ@n
  4. n1: Nam
  5. k: Nam
  6. y3: ρ@k
  7. i: Idx k
  8. y6: ρ@Fld k i
  9. y7: ρ@n1
  10. ∀i:Idx k. Reg(s) ⇒ y3.i ∈ ρ@Fld k i
  11. ∀i:Idx n1. Reg(s) ⇒ y7.i ∈ ρ@Fld n1 i
  12. ∀i@0:Idx (Fld k i)
        Reg(s) ⇒ y6.i@0 ∈ ρ@Fld (Fld k i) i@0
  ⊢ ∀i@0:Idx n1
  |    Reg(s) ⇒ [(y3:ρ@k).i:=y6@y7].i@0 ∈ ρ@Fld n1 i@0
  |
  BY D 0 THENM RecCaseSplit ‘ref‘ THENM SplitOnConclITE THEN A
  |  uto
  |\
  | 13. i@0: Idx n1
  | 14. ↑(y3=₀y7)
  | 15. ↑(i =꜀ i@0)
  | 16. Reg(s)
  | ⊢ y6 ∈ ρ@Fld n1 i@0
  | |
  1 BY InstLemma ‘par_type_lemma‘ [⌈m:Nam × Idx m⌉;⌈λ₂p.let <m
```

```
  | |   ,j> = p in ρ@Fld m j⌉;
  | |   ⌈<k, i>⌉;⌈<n1, i@0>⌉;⌈y6⌉] THENW Auto
  | |\
  | | ⊢ <k, i> = <n1, i@0>
  | | |
  1 2 BY EqCD THENA Auto
  | | |\
  | | | ⊢ k = n1
  | | | |
  1 2 3 BY FLemma `eq_obj_imp_eq_nam` [16;14] THEN Auto
  | |  \
  | |   ⊢ i = i@0
  | |   |
  1 2   BY D 16 THEN D 17 THEN InstHyp [⌈k⌉;⌈n1⌉;⌈i⌉;⌈i@0⌉] 18
  | |   |   THENM D 19 THEN Auto
  | |   |
  | |   16. ∀n,k:Nam.  ↑(n =_c k) ⟺ n = k
  | |   17. ∀n,k:Nam. ∀c:Cor n. ∀d:Cor k.
  | |         ↑(c =_c d) ⟺ (n = k) c∧ (c = d)
  | |   18. ∀n,k:Nam. ∀i:Idx n. ∀j:Idx k.
  | |         ↑(i =_c j) ⟺ (n = k) c∧ (i = j)
  | |   19. ↑(i =_c i@0) ⇒ (k = n1) c∧ (i = i@0)
  | |   20. ↑(i =_c i@0) ⇐ (k = n1) c∧ (i = i@0)
  | |   |
  1 2   BY RWH (HypC 18) 15 THENM D (15) THEN Auto

  |  \
  |   17. y6 ∈ let <m,j> = <n1, i@0> in ρ@Fld m j
  |   |
  1   BY Reduce 17 THEN Auto
   \
    13. i@0: Idx n1
    14. ¬↑(y3=_0y7) ∨ ¬↑(i =_c i@0)
    15. Reg(s)
    ⊢ y7.i@0 ∈ ρ@Fld n1 i@0
    |
    BY BHyp 11 THEN Auto

*A c_update    [cor(x):=t@y] ==  [core(x:ρ@nam(x)):=t@y]
*T c_update_wf

⊢ ∀s:Sign. ∀n,k:Nam. ∀x:ρ@k. ∀t:Cor k. ∀y:ρ@n.
|    [cor(x):=t@y] ∈ ρ@n
|
BY Unfold `c_update` 0 THEN Auto
|\
| 1. s: Sign
| 2. n: Nam
| 3. k: Nam
| 4. x: ρ@k
| 5. t: Cor k
| 6. y: ρ@n
| ⊢ x ∈ ρ@nam(x)
| |
```

```
1 BY BLemma ‘nam_sound_mem‘ THENA Auto
 \
  1. s: Sign
  2. n: Nam
  3. k: Nam
  4. x: ρ@k
  5. t: Cor k
  6. y: ρ@n
  ⊢ t ∈ Cor nam(x)
  |
  BY SubstClause ⌈Cor nam(x)⌉ 5 THENA EqCD THEN Auto
  |
  ⊢ k = nam(x)
  |
  BY RWH (LemmaC ‘nam_sound‘) 0 THEN Auto

*A r_update    [x.i:=z@y] ==  [(x:ρ@nam(x)).i:=z@y]
*T r_update_wf

⊢ ∀s:Sign. ∀n,k:Nam. ∀x:ρ@k. ∀i:Idx k. ∀z:ρ@Fld k i. ∀y:ρ@n.
|    [x.i:=z@y] ∈ ρ@n
|
BY Unfold ‘r_update‘ 0 THEN Auto
|\
| 1. s: Sign
| 2. n: Nam
| 3. k: Nam
| 4. x: ρ@k
| 5. i: Idx k
| 6. z: ρ@Fld k i
| 7. y: ρ@n
| ⊢ x ∈ ρ@nam(x)
| |
1 BY BLemma ‘nam_sound_mem‘ THEN Auto
|\
| 1. s: Sign
| 2. n: Nam
| 3. k: Nam
| 4. x: ρ@k
| 5. i: Idx k
| 6. z: ρ@Fld k i
| 7. y: ρ@n
| ⊢ i ∈ Idx nam(x)
| |
1 BY SubstClause ⌈Idx nam(x)⌉ 5 THEN Auto
| |
| ⊢ Idx k = Idx nam(x)
| |
1 BY EqCD THEN Auto
| |
| ⊢ k = nam(x)
| |
1 BY RWH (LemmaC ‘nam_sound‘) 0 THEN Auto
 \
```

```
  1. s: Sign
  2. n: Nam
  3. k: Nam
  4. x: ρ@k
  5. i: Idx k
  6. z: ρ@Fld k i
  7. y: ρ@n
  ⊢ z ∈ ρ@Fld nam(x) i
  |
  BY SubstClause ⌈ρ@Fld nam(x) i⌉ 6 THEN Auto
  |
  ⊢ ρ@Fld k i = ρ@Fld nam(x) i
  |
  BY RepeatMFor 3 EqCD THEN Auto
  |
  ⊢ k = nam(x)
  |
  BY RWH (LemmaC ‘nam_sound‘) 0 THEN Auto

*A get_ref                    get_ref(o;i;s) ==  o.i
*T get_ref_wf

⊢ ∀s:Sign. ∀n:Nam. ∀o:ρ@n. ∀i:Idx n.
|    Reg(s) ⇒ get_ref(o;i;s) ∈ ρ@Fld nam(o) i
|
BY Unfold ‘get_ref‘ 0 THEN Auto
|\
| 1. s: Sign
| 2. n: Nam
| 3. o: ρ@n
| 4. i: Idx n
| 5. Reg(s)
| ⊢ o ∈ ρ@nam(o)
| |
1 BY BLemma ‘nam_sound_mem‘ THEN Auto
 \
  1. s: Sign
  2. n: Nam
  3. o: ρ@n
  4. i: Idx n
  5. Reg(s)
  ⊢ i ∈ Idx nam(o)
  |
  BY SubstClause ⌈Idx nam(o)⌉ 4 THENA EqCD THEN Auto
  |
  ⊢ n = nam(o)
  |
  BY RWH (LemmaC ‘nam_sound‘) 0 THEN Auto

*T get_ref_wf_2

⊢ ∀s:Sign. ∀n:Nam. ∀o:ρ@n. ∀i:Idx n.
|    Reg(s) ⇒ get_ref(o;i;s) ∈ ρ@Fld n i
|
```

```
BY Auto
|
1. s: Sign
2. n: Nam
3. o: ρ@n
4. i: Idx n
5. Reg(s)
⊢ get_ref(o;i;s) ∈ ρ@Fld n i
|
BY SubstClause ⌈Idx nam(o)⌉ 4 THENA Auto
|\
| 4. i: Idx nam(o)
| |
1 BY InstLemma ‘par_type_lemma‘ [⌈n:Nam × Idx n⌉;⌈λ2ni.ρ@let <
| |  n,i> = ni in Fld n i⌉;
| |  ⌈<nam(o), i>⌉;⌈<n, i>⌉;⌈get_ref(o;i;s)⌉] THEN Reduce (-1)
| |   THEN Auto
| |\
| | ⊢ i ∈ Idx n
| | |
1 2 BY SubstClause ⌈Idx n⌉ 4 THEN Auto THEN EqCD THEN Auto
| | |
| | ⊢ nam(o) = n
| | |
1 2 BY RWH (LemmaC ‘nam_sound‘) 0 THEN Auto
| |\
| | ⊢ o ∈ ρ@nam(o)
| | |
1 2 BY BLemma ‘nam_sound_mem‘ THEN Auto
|  \
|   ⊢ <nam(o), i> = <n, i>
|   |
1   BY EqCD THEN Auto
|   |
|   ⊢ nam(o) = n
|   |
1   BY RWH (LemmaC ‘nam_sound‘) 0 THEN Auto
 \
  ⊢ Idx n = Idx nam(o)
  |
  BY EqCD THEN Auto
  |
  ⊢ n = nam(o)
  |
  BY RWH (LemmaC ‘nam_sound‘) 0 THEN Auto

*T comb_for_class_wf

⊢ (λs,n,z.ρ@n) ∈ s:Sign → n:Nam → ↓True → 𝕌
|
BY ProveOpCombTyping ‘class_wf‘

*C class_end                 ****************************
```

```
*C j_type_begin  ************ J_TYPE ************
*C This theory defines universe of names for J types.
*A j_class_name
            JClassName(f) ==  {a:Atom| ↑(f a)}
*T j_class_name_wf

⊢ ∀f:Atom → 𝔹. JClassName(f) ∈ 𝕌
|
BY Unfold ‘j_class_name‘ 0 THEN Auto

*A type        Type ==
                rec(T.Unit + Unit + Unit + JClassName(f) + T)
*T type_wf

⊢ ∀f:Atom → 𝔹. Type ∈ 𝕌
|
BY Unfold ‘type‘ 0 THEN Auto

*A bool_type                  boolean ==  inl ·
*T bool_type_wf

⊢ ∀f:Atom → 𝔹. boolean ∈ Type
|
BY Unfolds ‘‘bool_type type‘‘ 0 THEN UnivCD THENM MemTypeCD TH
   EN Auto

*A int_type                   int ==  inr (inl · )
*T int_type_wf

⊢ ∀f:Atom → 𝔹. int ∈ Type
|
BY Unfolds ‘‘int_type type‘‘ 0 THEN UnivCD THENM MemTypeCD THE
   N Auto

*A exc_type                   Exception ==  inr inr (inl · )
*T exc_type_wf

⊢ ∀f:Atom → 𝔹. Exception ∈ Type
|
BY Unfolds ‘‘exc_type type‘‘ 0 THEN UnivCD THENM MemTypeCD THE
   N Auto

*A class_type  Class(n) ==  inr inr inr (inl n )
*T class_type_wf

⊢ ∀f:Atom → 𝔹. ∀n:JClassName(f).  Class(n) ∈ Type
|
BY RepUnfolds ‘‘type class_type j_class_name‘‘ 0 THEN UnivCD T
   HENM MemTypeCD THEN A
   uto

*T comb_for_class_type_wf

⊢ (λf,n,z.Class(n)) ∈ f:(Atom → 𝔹)
|                       → n:JClassName(f)
|                       → ↓True
|                       → Type
|
BY ProveOpCombTyping ‘class_type_wf‘
```

```
*A array_type                 t[] ==  inr inr inr inr t
*T array_type_wf

⊢ ∀f:Atom → 𝔹. ∀t:Type.  t[] ∈ Type
|
BY Unfolds ‘‘array_type type‘‘ 0 THEN UnivCD THENM MemTypeCD T
   HEN Auto

*T comb_for_array_type_wf

⊢ (λf,t,z.t[]) ∈ f:(Atom → 𝔹) → t:Type → ↓True → Type
|
BY ProveOpCombTyping ‘array_type_wf‘

*A type_cases  case x
               of bool -> b
                | int -> z
                | Exception -> e
                | Class(n) -> c[n]
                | t[] -> a[t]
                ==
                 case x
                 of inl(x1) => b
                  | inr(x2) => case x2
                               of inl(x3) => z
                                | inr(x4) => case x4
                                             of inl(x5) => e
                                              | inr(x6) => cas
               e x6
                                                            of
               inl(n) => c[n]
                                                             |
               inr(t) => a[t]
*T type_cases_wf

⊢ ∀f:Atom → 𝔹. ∀T:𝕌. ∀x:Type. ∀b,z,e:T.
|  ∀c:JClassName(f) → T. ∀a:Type → T.
|    case x
|    of bool -> b
|     | int -> z
|     | Exception -> e
|     | Class(n) -> c[n]
|     | t[] -> a[t]
|     ∈ T
|
BY UnivCD THENM TypeHD 3 THENM Reduce 0 THEN Auto

*M eq_type_ml  (x =t y)
               ==r case x
                   of bool -> case y
                              of bool -> tt
                               | int -> ff
                               | Exception -> ff
                               | Class(n) -> ff
                               | t[] -> ff
```

```
                    | int -> case y
                            of bool -> ff
                             | int -> tt
                             | Exception -> ff
                             | Class(n) -> ff
                             | t[] -> ff

                    | Exception -> case y
                                  of bool -> ff
                                   | int -> ff
                                   | Exception -> tt
                                   | Class(n) -> ff
                                   | t[] -> ff

                    | Class(n) -> case y
                                 of bool -> ff
                                  | int -> ff
                                  | Exception -> ff
                                  | Class(m) -> n =a m
                                  | t[] -> ff

                    | t[] -> case y
                            of bool -> ff
                             | int -> ff
                             | Exception -> ff
                             | Class(n) -> ff
                             | s[] -> (t =t s)
```

```
*T eq_type_wf

⊢ ∀f:Atom → 𝔹. ∀x,y:Type.  (x =t y) ∈ 𝔹
|
BY D 0 THENM D 0 THENM TypeInd 2 THENM D 0 THENM TypeInd (-1)
|  THENM RecCaseSplit ‘e
|  q_type‘ THEN Auto
|\
| 1. f: Atom → 𝔹
| 2. x: JClassName(f)
| 3. x1: JClassName(f)
| ⊢ x ∈ Atom
| |
1 BY D 2 THEN Auto
|\
| 1. f: Atom → 𝔹
| 2. x: JClassName(f)
| 3. x1: JClassName(f)
| ⊢ x1 ∈ Atom
| |
1 BY D 3 THEN Auto
 \
  1. f: Atom → 𝔹
  2. y1: Type
  3. ∀y:Type. (y1 =t y) ∈ 𝔹
  4. y2: Type
  5. (y1[] =t y2) ∈ 𝔹
```

```
  ⊢ (y1 =t y2) ∈ 𝔹
  |
  BY BHyp 3 THEN Auto

*T comb_for_eq_type_wf

⊢ (λf,x,y,z.(x =t y)) ∈ f:(Atom → 𝔹)
|                        → x:Type
|                        → y:Type
|                        → ↓True
|                        → 𝔹
|
BY ProveOpCombTyping `eq_type_wf`

*T assert_of_eq_type

⊢ ∀f:Atom → 𝔹. ∀x,y:Type.  ↑(x =t y) ⇒ x = y
|
BY D 0 THENM D 0 THENM TypeInd 2 THENM D 0 THENM TypeInd (-1)
|  THENM RecCaseSplit `e
|  q_type` THEN Auto THEN Try (ApFunToHypEquands `z'
|  ⌈case z of bool -> 0  | int -> 1  | Exception -> 2  | Class
|  (n) -> 3  | a[] -> 4⌉
|   ⌈ℤ⌉
|   (-1) THEN Reduce (-1) THEN Auto) THEN Try (RWH (bool_to_pr
|  opC) (-1) THENM EqCD
|  THEN Auto) THEN Try (D 3 THEN D 2 THEN Auto)
|\
| 1. f: Atom → 𝔹
| 2. x: Atom
| 3. ↑(f x)
| 4. x1: Atom
| 5. ↑(f x1)
| 6. x = x1
| ⊢ x = x1
| |
1 BY MemTypeCD THEN Auto
 \
  1. f: Atom → 𝔹
  2. y1: Type
  3. ∀y:Type. ↑(y1 =t y) ⇒ y1 = y
  4. y2: Type
  5. ↑(y1[] =t y2) ⇒ y1[] = y2
  6. ↑(y1 =t y2)
  ⊢ y1 = y2
  |
  BY BHyp 3 THEN Auto

*T eq_type_refl

⊢ ∀f:Atom → 𝔹. ∀x:Type.  ↑(x =t x)
|
BY UnivCD THENM TypeInd 2 THENM RecCaseSplit `eq_type` THEN Au
|  to
|
```

```
1. f: Atom → 𝔹
2. x: JClassName(f)
⊢ ↑x =a x
|
BY RWH bool_to_propC 0 THEN D 2 THEN Auto

*T eq_type_iff_eq

⊢ ∀f:Atom → 𝔹. ∀x,y:Type.  ↑(x =t y) ⟺ x = y
|
BY GenUnivCD THENA Auto
|\
| 1. f: Atom → 𝔹
| 2. x: Type
| 3. y: Type
| 4. ↑(x =t y)
| ⊢ x = y
| |
1 BY BLemma ‘assert_of_eq_type‘ THEN Auto
 \
  1. f: Atom → 𝔹
  2. x: Type
  3. y: Type
  4. x = y
  ⊢ ↑(x =t y)
  |
  BY RWH (HypC 4) 0 THENM BLemma ‘eq_type_refl‘ THEN Auto

*A spec        Spec ==  JClassName(f) → Atom → ?Type
*C j_type_end                 ********************************
```

```
*C j_signature_begin **** J_SIGNATURE ****
*C This theory describes J type structure in terms
   of a reference type signature.
*A j_act_cor   JActCor(x) ==
                 case x
                 of bool -> 𝔹
                  | int -> ℤ
                  | Exception -> Atom
                  | Class(n) -> Void
                  | t[] -> ℕ

*T j_act_cor_wf

⊢ ∀f:Atom → 𝔹. ∀x:Type.  JActCor(x) ∈ 𝕌
|
BY Unfold ‘j_act_cor‘ 0 THEN Auto

*A j_eq_act_cor
               (a =ac b) ==
                 case x
                 of bool -> a =b b
                  | int -> (a =z b)
                  | Exception -> a =a b
                  | Class(n) -> tt
                  | t[] -> (a =z b)

*T j_eq_act_cor_wf

⊢ ∀f:Atom → 𝔹. ∀x:Type. ∀a,b:JActCor(x).  (a =ac b) ∈ 𝔹
|
BY UnivCD THENM TypeHD 2 THENM Reduce 0 THEN Auto

*T assert_of_j_eq_act_cor

⊢ ∀f:Atom → 𝔹. ∀x:Type. ∀a,b:JActCor(x).  ↑(a =ac b) ⇒ a = b
|
BY UnivCD THENM TypeHD 2 THENM Reduce 0 THENM RWH bool_to_prop
   C 0 THEN Auto

*A j_cor                      JCor(x) ==  ?JActCor(x)
*T j_cor_wf

⊢ ∀f:Atom → 𝔹. ∀x:Type.  JCor(x) ∈ 𝕌
|
BY Unfold ‘j_cor‘ 0 THEN Auto

*T comb_for_j_cor_wf

⊢ (λf,x,z.JCor(x)) ∈ f:(Atom → 𝔹) → x:Type → ↓True → 𝕌
|
BY ProveOpCombTyping ‘j_cor_wf‘

*A j_c                        Jc ==  inr ·
*T j_c_wf
```

```
⊢ ∀f:Atom → 𝔹. ∀x:Type.  Jc ∈ JCor(x)
|
BY Unfolds ``j_c j_cor`` 0 THEN Auto

*A j_ac                      Jac(a) ==  inl a
*T j_ac_wf

⊢ ∀f:Atom → 𝔹. ∀x:Type. ∀a:JActCor(x).  Jac(a) ∈ JCor(x)
|
BY Unfolds ``j_ac j_cor`` 0 THEN Auto

*T comb_for_j_ac_wf

⊢ (λf,x,a,z.Jac(a)) ∈ f:(Atom → 𝔹)
|                     → x:Type
|                     → a:JActCor(x)
|                     → ↓True
|                     → JCor(x)
|
BY ProveOpCombTyping `j_ac_wf`

*A j_cor_cases case x of Jc -> c  | Jac(a) -> ac[a] ==
                case x of inl(a) => ac[a] | inr(i) => c
*T j_cor_cases_wf

⊢ ∀f:Atom → 𝔹. ∀x:Type. ∀T:𝕌. ∀c:T. ∀ac:JActCor(x) → T.
|  ∀a:JCor(x).
|    case a of Jc -> c  | Jac(a) -> ac[a] ∈ T
|
BY UnivCD THENM JCorHD (-1) THENM Reduce 0 THEN Auto

*A j_eq_cor    (a =c b) ==
                case a
                of Jc -> case b of Jc -> tt  | Jac(u) -> ff
                 | Jac(ac) -> case b
                              of Jc -> ff
                               | Jac(bc) -> (ac =ac bc)

*T j_eq_cor_wf

⊢ ∀f:Atom → 𝔹. ∀x:Type. ∀a,b:JCor(x).  (a =c b) ∈ 𝔹
|
BY Unfold `j_eq_cor` 0 THEN Auto

*T comb_for_j_eq_cor_wf

⊢ (λf,x,a,b,z.(a =c b)) ∈ f:(Atom → 𝔹)
|                         → x:Type
|                         → a:JCor(x)
|                         → b:JCor(x)
|                         → ↓True
|                         → 𝔹
|
BY ProveOpCombTyping `j_eq_cor_wf`

*T assert_of_j_eq_cor
```

```
⊢ ∀f:Atom → 𝔹. ∀x:Type. ∀a,b:JCor(x).  ↑(a =c b) ⇒ a = b
|
BY Unfold ‘j_eq_cor‘ 0 THEN UnivCD THENM JCorHD (-2) THENM JCo
|  rHD (-1) THENM Reduce
|   0 THEN Auto
|
1. f: Atom → 𝔹
2. x: Type
3. a: JActCor(x)
4. a1: JActCor(x)
5. ↑(a =ac a1)
⊢ Jac(a) = Jac(a1)
|
BY EqCD THENM FLemma ‘assert_of_j_eq_act_cor‘ [5] THEN Auto

*T j_eq_cor_refl

⊢ ∀f:Atom → 𝔹. ∀x:Type. ∀a:JCor(x).  ↑(a =c a)
|
BY UnivCD THENM JCorHD 3 THENM Unfold ‘j_eq_cor‘ 0 THENM Reduc
|  e 0 THENA Auto
|\
| 1. f: Atom → 𝔹
| 2. x: Type
| 3. a: JActCor(x)
| ⊢ ↑(a =ac a)
| |
1 BY TypeHD 2 THEN Unfold ‘j_eq_act_cor‘ 0 THEN Reduce 0 THEN
|    RWH bool_to_propC 0 THE
|    N Auto
 \
  1. f: Atom → 𝔹
  2. x: Type
  ⊢ True
  |
  BY Auto

*T j_eq_cor_iff_eq

⊢ ∀f:Atom → 𝔹. ∀x:Type. ∀a,b:JCor(x).  ↑(a =c b) ⟺ a = b
|
BY GenUnivCD THENA Auto
|\
| 1. f: Atom → 𝔹
| 2. x: Type
| 3. a: JCor(x)
| 4. b: JCor(x)
| 5. ↑(a =c b)
| ⊢ a = b
| |
1 BY BLemma ‘assert_of_j_eq_cor‘ THEN Auto
 \
  1. f: Atom → 𝔹
  2. x: Type
```

```
  3. a: JCor(x)
  4. b: JCor(x)
  5. a = b
  ⊢ ↑(a =c b)
  |
  BY RWH (HypC 5) 0 THENM BLemma ‘j_eq_cor_refl‘ THEN Auto
```

```
*A j_idx       JIdx(x) ==
                 case x
                 of bool -> Void
                  | int -> Void
                  | Exception -> Void
                  | Class(n) -> {a:Atom| ↑isl(s n a)}
                  | t[] -> ℕ
```

```
*T j_idx_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀x:Type.  JIdx(x) ∈ 𝕌
|
BY Unfold ‘j_idx‘ 0 THEN Auto
```

```
*T comb_for_j_idx_wf

⊢ (λf,s,x,z.JIdx(x)) ∈ f:(Atom → 𝔹)
|                      → s:Spec
|                      → x:Type
|                      → ↓True
|                      → 𝕌
|
BY ProveOpCombTyping ‘j_idx_wf‘
```

```
*A j_eq_idx    (i =i j) ==
                 case x
                 of bool -> tt
                  | int -> tt
                  | Exception -> tt
                  | Class(n) -> i =a j
                  | t[] -> (i =z j)
```

```
*T j_eq_idx_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀x:Type. ∀i,j:JIdx(x).  (i =i j) ∈ 𝔹
|
BY UnivCD THENM TypeHD 3 THENM Reduce 0 THEN Auto
```

```
*T comb_for_j_eq_idx_wf

⊢ (λf,s,x,i,j,z.(i =i j)) ∈ f:(Atom → 𝔹)
|                             → s:Spec
|                             → x:Type
|                             → i:JIdx(x)
|                             → j:JIdx(x)
|                             → ↓True
|                             → 𝔹
|
BY ProveOpCombTyping ‘j_eq_idx_wf‘
```

```
*T assert_j_eq_idx

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀x:Type. ∀i,j:JIdx(x).
|    ↑(i =i j) ⇒ i = j
|
BY UnivCD THENM TypeHD 3 THENM Reduce 0 THENM RWH (bool_to_pro
|  pC) 0 THEN Auto
|
1. f: Atom → 𝔹
2. s: Spec
3. x: Type
4. n: JClassName(f)
5. x = Class(n)
6. i: {a:Atom| ↑isl(s n a)}
7. j: {a:Atom| ↑isl(s n a)}
8. i = j
⊢ i = j
|
BY D 7 THEN D 6 THEN MemTypeCD THEN Auto

*T j_eq_idx_refl

⊢ ∀f:Atom → 𝔹. ∀x:Type. ∀s:Spec. ∀i:JIdx(x).  ↑(i =i i)
|
BY UnivCD THENM TypeHD 2 THENM Reduce 0 THENM RWH bool_to_prop
   C 0 THEN Auto

*T j_eq_idx_iff_eq

⊢ ∀f:Atom → 𝔹. ∀x:Type. ∀s:Spec. ∀i,j:JIdx(x).
|    ↑(i =i j) ⟺ i = j
|
BY GenUnivCD THENA Auto
|\
| 1. f: Atom → 𝔹
| 2. x: Type
| 3. s: Spec
| 4. i: JIdx(x)
| 5. j: JIdx(x)
| 6. ↑(i =i j)
| ⊢ i = j
| |
1 BY BLemma ‘assert_j_eq_idx‘ THEN Auto
 \
  1. f: Atom → 𝔹
  2. x: Type
  3. s: Spec
  4. i: JIdx(x)
  5. j: JIdx(x)
  6. i = j
  ⊢ ↑(i =i j)
  |
  BY RWH (HypC 6) 0 THENM BLemma ‘j_eq_idx_refl‘ THEN Auto
```

```
*A j_fld       Jfld(x.i) ==
                 case x
                 of bool -> boolean
                  | int -> int
                  | Exception -> Exception
                  | Class(n) -> outl(s n i)
                  | t[] -> t

*T j_fld_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀x:Type. ∀i:JIdx(x).
|    Jfld(x.i) ∈ Type
|
BY UnivCD THENM TypeHD 3 THEN Reduce 0 THEN Auto
|
1. f: Atom → 𝔹
2. s: Spec
3. x: Type
4. n: JClassName(f)
5. x = Class(n)
6. i: {a:Atom| ↑isl(s n a)}
⊢ ↑isl(s n i)
|
BY D 6 THEN Unhide THEN Auto

*A j_sign      JSign(f;s) ==
                 <Type
                 , λt.JCor(t)
                 , λt.Jc
                 , λt.JIdx(t)
                 , λt,i.Jfld(t.i)
                 , λt1,t2.(t1 =t t2)
                 , λt1,c1,t2,c2.
                    if (t1 =t t2) then (c1 =c c2) else ff fi
                 , λt1,i1,t2,i2.
                    if (t1 =t t2) then (i1 =i i2) else ff fi >
*T j_sign_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec.  JSign(f;s) ∈ Sign
|
BY Unfold `j_sign` 0 THEN UnivCD THENA Auto
|
1. f: Atom → 𝔹
2. s: Spec
⊢ <Type
|  , λt.JCor(t)
|  , λt.Jc
|  , λt.JIdx(t)
|  , λt,i.Jfld(t.i)
|  , λt1,t2.(t1 =t t2)
|  , λt1,c1,t2,c2.if (t1 =t t2) then (c1 =c c2) else ff fi
|  , λt1,i1,t2,i2.if (t1 =t t2) then (i1 =i i2) else ff fi >
|  ∈ Sign
|
```

```
BY Unfold `signature` 0 THEN RepeatMFor 11 (MemCD THEN Reduce
|  0) THENM Try (SplitOn
|  ConclITE) THEN Auto
|\
| 3. t1: Type
| 4. c1: JCor(t1)
| 5. t2: Type
| 6. c2: JCor(t2)
| 7. ↑(t1 =t t2)
| ⊢ c2 ∈ JCor(t1)
| |
1 BY FLemma `assert_of_eq_type` [7] THENM SubstClause ⌜JCor(t1
|    )⌝
|     6 THEN Try EqCD THEN Auto
 \
  3. t1: Type
  4. i1: JIdx(t1)
  5. t2: Type
  6. i2: JIdx(t2)
  7. ↑(t1 =t t2)
  ⊢ i2 ∈ JIdx(t1)
  |
  BY FLemma `assert_of_eq_type` [7] THENM SubstClause ⌜JIdx(t1
     )⌝
      6 THEN Try EqCD THEN Auto
```

```
*T j_sign_reg

⊢ ∀f:Atom → 𝔹. ∀s:Spec.  Reg(JSign(f;s))
|
BY RepUnfolds ``reg_sign eq_nam eq_cor eq_idx`` 0 THEN Reduce
|  0 THEN UnivCD THENM (
|  D 0 THENL [Id; D 0]) THENM UnivCD THENA Auto
|\
| 1. f: Atom → 𝔹
| 2. s: Spec
| 3. n: Type
| 4. k: Type
| ⊢ ↑(n =t k) ⟺ n = k
| |
1 BY RWH (LemmaC `eq_type_iff_eq`) 0 THEN Auto
|\
| 1. f: Atom → 𝔹
| 2. s: Spec
| 3. n: Type
| 4. k: Type
| 5. c: JCor(n)
| 6. d: JCor(k)
| ⊢ ↑if (n =t k) then (c =c d) else ff fi
| |  ⟺ (n = k) c∧ (c = d)
| |
1 BY SplitOnConclITE THENM RWH (LemmaC `eq_type_iff_eq`) (-1)
| |  THENA Auto
| |\
```

```
| | 7. n = k
| | ⊢ ↑(c =c d) ⟺ (n = k) c∧ (c = d)
| | |
1 2 BY SubstClause ⌈JCor(n)⌉ 6 THENA (EqCD THEN Auto)
| | |
| | 6. d: JCor(n)
| | |
1 2 BY RWH (LemmaC ‘j_eq_cor_iff_eq‘) 0 THEN Auto THEN D 8 THE
| |    N Auto
|  \
|   7. ¬(n = k)
|   ⊢ ↑ff ⟺ (n = k) c∧ (c = d)
|   |
1   BY Reduce 0 THEN Auto THEN D 8 THEN Auto
 \
  1. f: Atom → 𝔹
  2. s: Spec
  3. n: Type
  4. k: Type
  5. i: JIdx(n)
  6. j: JIdx(k)
  ⊢ ↑if (n =t k) then (i =i j) else ff fi
  |  ⟺ (n = k) c∧ (i = j)
  |
  BY SplitOnConclITE THENM RWH (LemmaC ‘eq_type_iff_eq‘) (-1)
  |  THENA Auto
  |\
  | 7. n = k
  | ⊢ ↑(i =i j) ⟺ (n = k) c∧ (i = j)
  | |
  1 BY SubstClause ⌈JIdx(n)⌉ 6 THENA (EqCD THEN Auto)
  | |
  | 6. j: JIdx(n)
  | |
  1 BY RWH (LemmaC ‘j_eq_idx_iff_eq‘) 0 THEN Auto THEN D 8 THE
  |    N Auto
   \
    7. ¬(n = k)
    ⊢ ↑ff ⟺ (n = k) c∧ (i = j)
    |
    BY Reduce 0 THEN Auto THEN D 8 THEN Auto

*C j_signature_end
               *************************************
```

```
*C j_class_begin **** J_CLASS ****
*C This theory provides type theoretical model for J
   type structure using the general notion of reference type.
*A j_class                  J(t) ==  ρ@t
*T j_class_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀t:Type.  J(t) ∈ 𝕌
|
BY Unfold ‘j_class‘ 0 THEN Auto

*A j_nil_obj                t₀ ==  ⊕(t)
*T j_nil_obj_wf

⊢ ∀f:Atom → 𝔹. ∀t:Type. ∀s:Spec.  t₀ ∈ J(t)
|
BY Unfolds ‘‘j_nil_obj j_class‘‘ 0 THEN Auto

*A j_c_update               [cor(x):=c@y] ==  [cor(x):=c@y]
*T j_c_update_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀t,r:Type. ∀x:J(t). ∀c:JCor(t).
|  ∀y:J(r).
|    [cor(x):=c@y] ∈ J(r)
|
BY Unfolds ‘‘j_class j_c_update‘‘ 0 THEN Auto

*A j_r_update               [x.i:=z@y] ==  [x.i:=z@y]
*T j_r_update_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀t,r:Type. ∀x:J(t). ∀i:JIdx(t).
|  ∀z:J(Jfld(t.i)). ∀y:J(r).
|    [x.i:=z@y] ∈ J(r)
|
BY Unfolds ‘‘j_class j_r_update‘‘ 0 THEN Auto

*A j_cor_const JCorConst(c;t) ==  [cor(t₀):=c@t₀]
*T j_cor_const_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀t:Type. ∀c:JCor(t).
|    JCorConst(c;t) ∈ J(t)
|
BY Unfold ‘j_cor_const‘ 0 THEN Auto

*A j_const     JConst(c;t) ==  JCorConst(Jac(c);t)
*T j_const_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀t:Type. ∀c:JActCor(t).
|    JConst(c;t) ∈ J(t)
|
BY Unfold ‘j_const‘ 0 THEN Auto

*A j_cor_oper2 j_cor_oper2(u,v.op[u; v];x;y) ==
                 case x
                 of Jc -> Jc
                  | Jac(ax) -> case y
                               of Jc -> Jc
                                | Jac(ay) -> Jac(op[ax; ay])
```

```
*T j_cor_oper2_wf

⊢ ∀f:Atom → 𝔹. ∀t:Type. ∀op:JActCor(t)
|                            → JActCor(t)
|                            → JActCor(t). ∀x,y:JCor(t).
|    j_cor_oper2(u,v.op[u;v];x;y) ∈ JCor(t)
|
BY Unfold `j_cor_oper2` 0 THEN Auto

*A get_j_cor                 get_j_cor(x;f;s) ==  cor(x)
*T get_j_cor_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀t:Type. ∀o:J(t).
|    get_j_cor(o;f;s) ∈ JCor(t)
|
BY Unfold `get_j_cor` 0 THEN Auto
|
1. f: Atom → 𝔹
2. s: Spec
3. t: Type
4. o: J(t)
⊢ cor(o) ∈ JCor(t)
|
BY InstLemma `cor_wf` [⌈JSign(f;s)⌉;⌈t⌉;⌈o⌉] THENA Auto
|\
| ⊢ o ∈ ρ@t
| |
1 BY Unfold `j_class` 4 THEN Auto
|\
| ⊢ Reg(JSign(f;s))
| |
1 BY BLemma `j_sign_reg` THEN Auto
 \
  5. cor(o) ∈ Cor t
  |
  BY Reduce 5 THEN Auto

*A simple_obj_oper2
               simple_obj_oper2(u,v.op[u; v];x;y;f;s;t) ==
                 JCorConst(j_cor_oper2(u,v.op[u; v];get_j_cor(
               x;f;s);get_j_cor(x;f;s));t)
*T simple_obj_oper2_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀t:Type. ∀x,y:J(t).
|  ∀op:JActCor(t) → JActCor(t) → JActCor(t).
|    simple_obj_oper2(u,v.op[u;v];x;y;f;s;t) ∈ J(t)
|
BY Unfold `simple_obj_oper2` 0 THEN Auto

*A get_type                  get_type(x) ==  nam(x)
*T get_type_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀t:Type. ∀x:J(t).  get_type(x) ∈ Type
|
BY Unfold `get_type` 0 THEN Auto
```

```
|
1. f: Atom → 𝔹
2. s: Spec
3. t: Type
4. x: J(t)
⊢ nam(x) ∈ Type
|
BY InstLemma ‘nam_wf‘ [⌈JSign(f;s)⌉;⌈t⌉;⌈x⌉] THENA Auto
|\
| ⊢ x ∈ ρ@t
| |
1 BY Unfold ‘j_class‘ 4 THEN Auto
 \
  5. nam(x) ∈ Nam
  |
  BY Reduce 5 THEN Auto

*T get_type_sound

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀t:Type. ∀x:J(t).  get_type(x) = t
|
BY Unfold ‘get_type‘ 0 THEN Auto
|
1. f: Atom → 𝔹
2. s: Spec
3. t: Type
4. x: J(t)
⊢ nam(x) = t
|
BY InstLemma ‘nam_sound‘ [⌈JSign(f;s)⌉;⌈t⌉;⌈x⌉] THEN Auto
|\
| ⊢ x ∈ ρ@t
| |
1 BY Fold ‘j_class‘ 0 THEN Auto
 \
  5. nam(x) = t
  |
  BY Reduce 5 THEN Auto

*T get_type_sound_mem

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀t:Type. ∀x:J(t).  x ∈ J(get_type(x))
|
BY UnivCD THENA Auto
|
1. f: Atom → 𝔹
2. s: Spec
3. t: Type
4. x: J(t)
⊢ x ∈ J(get_type(x))
|
BY InstLemma ‘nam_sound_mem‘ [⌈JSign(f;s)⌉;⌈t⌉;⌈x⌉] THENA Auto
|\
| ⊢ x ∈ ρ@t
```

```
| |
1 BY Unfold `j_class` 4 THEN Auto
 \
  5. x ∈ ρ@nam(x)
  |
  BY Unfolds ``get_type j_class`` 0 THEN Auto

*A get_j_ref   (x.i) ==  get_ref(x;i;JSign(f;s))
*T get_j_ref_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀t:Type. ∀x:J(t). ∀i:JIdx(t).
|    (x.i) ∈ J(Jfld(t.i))
|
BY Unfolds ``get_j_ref j_class`` 0 THEN Auto
|
1. f: Atom → 𝔹
2. s: Spec
3. t: Type
4. x: ρ@t
5. i: JIdx(t)
⊢ Reg(JSign(f;s))
|
BY BLemma `j_sign_reg` THEN Auto

*A eq_j                       eq_j(x;y;f;s) ==  (x=₀y)
*T eq_j_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀t,r:Type. ∀x:J(t). ∀y:J(r).
|    eq_j(x;y;f;s) ∈ 𝔹
|
BY Unfolds ``eq_j j_class`` 0 THEN Auto

*C j_class_end ********************************
```

```
*C env_begin  ************ ENV ************
*C This theory describes environment
   or state of J abstract machine.
*A env                     Env ==  Atom → t:Type → J(t)
*A env_update  en{(v:tv) -> o} ==
                λw,tw.
                 if v =a w ∧b (tv =t tw)
                 then o
                 else en w tw
                 fi
*T env_update_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀en:Env. ∀v:Atom. ∀t:Type. ∀o:J(t).
|    en{(v:t) -> o} ∈ Env
|
BY UnivCD THENA Auto
|
1. f: Atom → 𝔹
2. s: Spec
3. en: Env
4. v: Atom
5. t: Type
6. o: J(t)
⊢ en{(v:t) -> o} ∈ Env
|
BY Unfold ‘env_update‘ 0 THEN RepeatMFor 2 MemCD THENA Auto
|
7. w: Atom
8. tw: Type
⊢ if v =a w ∧b (t =t tw) then o else en w tw fi  ∈ J(tw)
|
BY SplitOnConclITE THEN Auto
|
9. v = w
10. ↑(t =t tw)
⊢ o ∈ J(tw)
|
BY FLemma ‘assert_of_eq_type‘ [10] THENA Auto
|
11. t = tw
|
BY InstLemma ‘par_type_lemma‘ [⌈Type⌉;⌈λ2t.J(t)⌉;⌈t⌉;⌈tw⌉;⌈o⌉]
    THEN Auto

*C env_end     ********************************
```

```
*C exp_begin           ************ EXP ************
*C This theory defines Nuprl type representing
   J expressions and provides examples of  λ-terms,
   corresponding to some J expressions.
*A exp_value   ExpValue(t) ==  Env × (J(Exception) + J(t))
*T exp_value_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀t:Type.  ExpValue(t) ∈ 𝕌
|
BY Unfold ‘exp_value‘ 0 THEN Auto

*T exp_value_total

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀t:Type.  ExpValue(t) total
|
BY Unfold ‘exp_value‘ 0 THEN Auto

*A exp_value_exc
            ExpValueExc(en;ex) ==  <en, inl ex >
*T exp_value_exc_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀t:Type. ∀en:Env. ∀ex:J(Exception).
|    ExpValueExc(en;ex) ∈ ExpValue(t)
|
BY Unfolds ‘‘exp_value exp_value_exc‘‘ 0 THEN Auto

*A exp_value_make
            ExpValueMake(en;x) ==  <en, inr x >
*T exp_value_make_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀en:Env. ∀t:Type. ∀x:J(t).
|    ExpValueMake(en;x) ∈ ExpValue(t)
|
BY Unfolds ‘‘exp_value_make exp_value‘‘ 0 THEN Auto

*T exp_value_make_inj

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀en1,en2:Env. ∀t:Type. ∀o1,o2:J(t).
|    ExpValueMake(en1;o1) = ExpValueMake(en2;o2) ⇒ en1 = en2
|
BY Unfolds ‘‘exp_value exp_value_make‘‘ 0 THEN Auto
|
1. f: Atom → 𝔹
2. s: Spec
3. en1: Env
4. en2: Env
5. t: Type
6. o1: J(t)
7. o2: J(t)
8. <en1, inr o1 > = <en2, inr o2 >
⊢ en1 = en2
|
BY EqHD 8 THENM Reduce 8 THEN Auto
```

```
*A exp_value_cases
               case ev
               of Exc(en1,ex) -> exc[en1; ex]
                | Make(en2,x) -> mk[en2; x]
                ==
                 let <en,et> = ev
                 in
                 case et
                 of inl(ex) => exc[en; ex]
                  | inr(x) => mk[en; x]
*T exp_value_cases_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀t:Type. ∀T:𝕌. ∀exc:Env
|                                             → J(Exception)
|                                             → T. ∀mk:Env
|                                                       → J(t)
|                                                       → T.
|  ∀ev:ExpValue(t).
|    case ev
|    of Exc(en,ex) -> exc[en;ex]
|     | Make(en,x) -> mk[en;x]
|     ∈ T
|
BY Unfolds ‘‘exp_value exp_value_cases‘‘ 0 THEN Auto

*T exp_value_cases_wf_bar

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀t,t’:Type.
|  ∀exc:Env → J(Exception) → bar(ExpValue(t’)).
|  ∀mk:Env → J(t) → bar(ExpValue(t’)). ∀ev:bar(ExpValue(t)).
|    case ev
|    of Exc(en,ex) -> exc[en;ex]
|     | Make(en,x) -> mk[en;x]
|     ∈ bar(ExpValue(t’))
|
BY Unfolds ‘‘exp_value exp_value_cases‘‘ 0 THEN Auto

*T exp_value_cases_wf_bar2

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀t:Type. ∀exc:Env
|                                      → J(Exception)
|                                      → bar(StmtValue(f;s)).
|  ∀mk:Env → J(t) → bar(StmtValue(f;s)).
|  ∀ev:bar(ExpValue(t)).
|    case ev
|    of Exc(en,ex) -> exc[en;ex]
|     | Make(en,x) -> mk[en;x]
|     ∈ bar(StmtValue(f;s))
|
BY Unfolds ‘‘exp_value exp_value_cases‘‘ 0 THEN Auto

*A exp         Exp(t) ==  Env → bar(ExpValue(t))
*A init        init t ==  λen.ExpValueMake(en;t0)
*T init_wf
```

```
⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀t:Type.  init t ∈ Exp(t)
|
BY Unfold ‘init‘ 0 THEN Auto

*C init_com    "init" is similar too "new" constructor but it
               returns the same object each time when it appli
               ed.
*A const_exp   const_exp(c;t) ==
                 λen.ExpValueMake(en;JConst(c;t))
*T const_exp_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀t:Type. ∀c:JActCor(t).
|    const_exp(c;t) ∈ Exp(t)
|
BY Unfold ‘const_exp‘ 0 THEN Auto

*A int_const                (z) ==  const_exp(z;int)
*T int_const_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀z:ℤ.  (z) ∈ Exp(int)
|
BY Unfold ‘int_const‘ 0 THEN Auto

*A bool_const               (b) ==  const_exp(b;boolean)
*T bool_const_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀b:𝔹.  (b) ∈ Exp(boolean)
|
BY Unfold ‘bool_const‘ 0 THEN Auto

*A exc_const                (a) ==  const_exp(a;Exception)
*T exc_const_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀a:Atom.  (a) ∈ Exp(Exception)
|
BY Unfold ‘exc_const‘ 0 THEN Auto

*A simple_exp2 simple_exp2(u,v.op[u; v];x;y;f;s;t) ==
                λen.case x en
                    of Exc(en’,ex) -> ExpValueExc(en’;ex)
                     | Make(en’,vx) -> case y en’
                                       of Exc(en’’,ex) -> Exp
               ValueExc(en’’;ex)
                                        | Make(en’’,vy) -> Ex
               pValueMake(en’’;simple_obj_oper2(u,v.op[u; v];v
               x;vy;f;s;t))

*T simple_exp2_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀t:Type. ∀op:JActCor(t)
|                                     → JActCor(t)
|                                     → JActCor(t).
|  ∀x,y:Exp(t).
|    simple_exp2(u,v.op[u;v];x;y;f;s;t) ∈ Exp(t)
|
BY Unfold ‘simple_exp2‘ 0 THEN Auto
```

```
*A add_exp     (x + y) ==  simple_exp2(u,v.u + v;x;y;f;s;int)
*T add_exp_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀x,y:Exp(int).  (x + y) ∈ Exp(int)
|
BY Unfold `add_exp` 0 THEN Auto THEN Reduce 0 THEN Auto

*A sub_exp     (x - y) ==  simple_exp2(u,v.u - v;x;y;f;s;int)
*T sub_exp_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀x,y:Exp(int).  (x - y) ∈ Exp(int)
|
BY Unfold `sub_exp` 0 THEN Auto THEN Reduce 0 THEN Auto

*A mul_exp     (x × y) ==  simple_exp2(u,v.u * v;x;y;f;s;int)
*T mul_exp_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀x,y:Exp(int).  (x × y) ∈ Exp(int)
|
BY Unfold `mul_exp` 0 THEN Auto THEN Reduce 0 THEN Auto

*A and_exp     (x & y) ==
                 simple_exp2(u,v.u ∧b v;x;y;f;s;boolean)
*T and_exp_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀x,y:Exp(boolean).
|    (x & y) ∈ Exp(boolean)
|
BY Unfold `and_exp` 0 THEN Auto THEN Reduce 0 THEN Auto

*A ref_exp     (x.i) ==
                 λen.case x en
                     of Exc(en',ex) -> ExpValueExc(en';ex)
                      | Make(en',vx) -> ExpValueMake(en';(vx.i
               ))

*T ref_exp_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀t:Type. ∀x:Exp(t). ∀i:JIdx(t).
|    (x.i) ∈ Exp(Jfld(t.i))
|
BY Unfold `ref_exp` 0 THEN Auto

*A var_exp     (v:t) ==  λen.ExpValueMake(en;en v t)
*T var_exp_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀t:Type. ∀v:Atom.  (v:t) ∈ Exp(t)
|
BY Unfold `var_exp` 0 THEN Auto

*C exp_end     ********************************
```

```
*C stmt_begin      ************ STMT ************
*C This theory is similar to the theory exp except for that
   it deals with value-less expressions.
*A stmt_value  StmtValue(f;s) ==  Env × ?J(Exception)
*T stmt_value_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec.  StmtValue(f;s) ∈ 𝕌
|
BY Unfold ‘stmt_value‘ 0 THEN Auto

*T stmt_value_total

⊢ ∀f:Atom → 𝔹. ∀s:Spec.  StmtValue(f;s) total
|
BY Unfold ‘stmt_value‘ 0 THEN Auto

*A stmt_value_exc
            StmtValueExc(en;ex) ==  <en, inl ex >
*T stmt_value_exc_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀en:Env. ∀ex:J(Exception).
|    StmtValueExc(en;ex) ∈ StmtValue(f;s)
|
BY Unfolds ‘‘stmt_value stmt_value_exc‘‘ 0 THEN Auto

*A stmt_value_norm
            StmtValueNorm(en) ==  <en, inr · >
*T stmt_value_norm_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀en:Env.
|    StmtValueNorm(en) ∈ StmtValue(f;s)
|
BY Unfolds ‘‘stmt_value stmt_value_norm‘‘ 0 THEN Auto

*A stmt_value_cases
            case stv
            of StmtValueExc(en,ex) -> E[en; ex]
             | StmtValueNorm(en) -> N[en]
             ==
              let <en,r> = stv
              in
              case r
              of inl(ex) => E[en; ex]
               | inr(i) => N[en]
*T stmt_value_cases_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀stv:StmtValue(f;s). ∀T:𝕌.
|  ∀E:Env → J(Exception) → T. ∀N:Env → T.
|    case stv
|    of StmtValueExc(en,ex) -> E[en;ex]
|     | StmtValueNorm(en) -> N[en]
|     ∈ T
|
BY Unfolds ‘‘stmt_value_cases stmt_value‘‘ 0 THEN Auto

*T stmt_value_cases_wf_bar
```

```
⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀stv:bar(StmtValue(f;s)).
|  ∀E:Env → J(Exception) → bar(StmtValue(f;s)).
|  ∀N:Env → bar(StmtValue(f;s)).
|    case stv
|    of StmtValueExc(en,ex) -> E[en;ex]
|     | StmtValueNorm(en) -> N[en]
|     ∈ bar(StmtValue(f;s))
|
BY Unfolds ‘‘stmt_value stmt_value_cases‘‘ 0 THEN Auto

*T comb_for_stmt_value_cases_wf

⊢ (λf,s,stv,T,E,N,z.
|    case stv
|    of StmtValueExc(en,ex) -> E[en;ex]
|     | StmtValueNorm(en) -> N[en]
|    ) ∈ f:(Atom → 𝔹)
|        → s:Spec
|        → stv:StmtValue(f;s)
|        → T:𝕌
|        → E:(Env → J(Exception) → T)
|        → N:(Env → T)
|        → ↓True
|        → T
|
BY ProveOpCombTyping ‘stmt_value_cases_wf‘

*T stmt_value_cases_norm

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀stv:StmtValue(f;s). ∀T:𝕌.
|  ∀E:Env → J(Exception) → T. ∀N:Env → T.
|    ↑IsNorm(stv)
|    ⇒ case stv
|       of StmtValueExc(en,ex) -> E[en;ex]
|        | StmtValueNorm(en) -> N[en]
|
|       = N[env(stv)]
|
BY UnivCD THENA Auto
|
1. f: Atom → 𝔹
2. s: Spec
3. stv: StmtValue(f;s)
4. T: 𝕌
5. E: Env → J(Exception) → T
6. N: Env → T
7. ↑IsNorm(stv)
⊢ case stv
|  of StmtValueExc(en,ex) -> E[en;ex]
|   | StmtValueNorm(en) -> N[en]
|
|  = N[env(stv)]
|
BY StmtValueHD 3 THEN Reduce 0 THEN Auto
```

```
*T stmt_value_cases_exc

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀stv:StmtValue(f;s). ∀T:𝕌.
|  ∀E,N:Env → T.
|    ¬↑IsNorm(stv)
|    ⇒ case stv
|       of StmtValueExc(en,ex) -> E[en]
|        | StmtValueNorm(en) -> N[en]
|
|       = E[env(stv)]
|
BY UnivCD THENA Auto
|
1. f: Atom → 𝔹
2. s: Spec
3. stv: StmtValue(f;s)
4. T: 𝕌
5. E: Env → T
6. N: Env → T
7. ¬↑IsNorm(stv)
⊢ case stv
|  of StmtValueExc(en,ex) -> E[en]
|   | StmtValueNorm(en) -> N[en]
|
|  = E[env(stv)]
|
BY StmtValueHD 3 THEN Reduce 0 THEN Auto
|
3. en: Env
7. ¬True
⊢ N[en] = E[en]
|
BY D 7 THEN Auto

*A stmt_value_env
                env(v) ==
                  case v
                  of StmtValueExc(en,ex) -> en
                   | StmtValueNorm(en) -> en

*T stmt_value_env_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀v:StmtValue(f;s).  env(v) ∈ Env
|
BY Unfold `stmt_value_env` 0 THEN Auto

*A stmt_value_is_norm
                IsNorm(v) ==
                  case v
                  of StmtValueExc(en,ex) -> ff
                   | StmtValueNorm(en) -> tt

*T stmt_value_is_norm_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀v:StmtValue(f;s).  IsNorm(v) ∈ 𝔹
|
BY Unfold `stmt_value_is_norm` 0 THEN Auto
```

```
*T stmt_value_is_norm_wf_bar

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀v:bar(StmtValue(f;s)).
|    IsNorm(v) ∈ bar(𝔹)
|
BY RepUnfolds ``stmt_value_is_norm stmt_value_cases stmt_value
   `` 0 THEN Auto

*T stmt_value_is_norm_wf_bar2

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀v:bar(StmtValue(f;s)).
|    v in! StmtValue(f;s) ⇒ IsNorm(v) ∈ 𝔹
|
BY UnivCD THENM InstLemma `stmt_value_is_norm_wf` [⌈f⌉;⌈s⌉;⌈v⌉
   ] THEN AutoT

*A stmt        Stmt ==  Env → bar(StmtValue(f;s))
*A skip                       skip ==  λen.StmtValueNorm(en)
*T skip_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec.  skip ∈ Stmt
|
BY Unfold `skip` 0 THEN Auto

*A seq         st1; st2 ==
                 λen.case st1 en
                     of StmtValueExc(en',ex) -> st1 en
                      | StmtValueNorm(en') -> st2 en'

*T seq_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀st1,st2:Stmt.  st1; st2 ∈ Stmt
|
BY Unfold `seq` 0 THEN Auto

*A throw       throw(e) ==
                 λen.case e en
                     of Exc(en',ex) -> StmtValueExc(en';ex)
                      | Make(en',v) -> StmtValueExc(en';v)

*T throw_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀e:Exp(Exception).  throw(e) ∈ Stmt
|
BY Unfolds ``throw exp`` 0 THEN Auto

*A if          if(b;c;d;f;s) ==
                 λen.case b en
                     of Exc(en',ex) -> StmtValueExc(en';ex)
                      | Make(en',bv) -> case get_j_cor(bv;f;s)

                                         of Jc -> StmtValueExc(
               en';JConst("NilPointerException";Exception))
                                          | Jac(bav) -> if bav
                                         then c en'
                                         else d en'
                                         fi
```

```
*T if_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀b:Exp(boolean). ∀c,d:Stmt.
|    if(b;c;d;f;s) ∈ Stmt
|
BY Unfolds ``exp if`` 0 THEN UnivCD THENM MemCD THENM MemCD TH
|  ENA Auto
|\
| 1. f: Atom → 𝔹
| 2. s: Spec
| 3. b: Env → bar(ExpValue(boolean))
| 4. c: Stmt
| 5. d: Stmt
| 6. en: Env
| 7. en': Env
| 8. ex: J(Exception)
| ⊢ StmtValueExc(en';ex) ∈ bar(StmtValue(f;s))
| |
1 BY Auto
|\
| 1. f: Atom → 𝔹
| 2. s: Spec
| 3. b: Env → bar(ExpValue(boolean))
| 4. c: Stmt
| 5. d: Stmt
| 6. en: Env
| 7. en': Env
| 8. bv: J(boolean)
| ⊢ case get_j_cor(bv;f;s)
| |  of Jc -> StmtValueExc(en';JConst("NilPointerException";Exc
| |
| |  eption))
| |   | Jac(bav) -> if bav then c en' else d en' fi
| |   ∈ bar(StmtValue(f;s))
| |
1 BY InstLemma `j_cor_cases_wf` [⌈f⌉;⌈boolean⌉;⌈bar(StmtValue(
| |  f;s))⌉;
| |  ⌈StmtValueExc(en';JConst("NilPointerException";Exception)
| |  )⌉;⌈λ₂bav.if bav
| |
| |      then c en'
| |
| |      else d en'
| |
| |      fi ⌉;
| |  ⌈get_j_cor(bv;f;s)⌉] THEN Auto
| |
| ⊢ JActCor(boolean) ∈ 𝕌{18}
| |
1 BY Reduce 0 THEN Auto
 \
  1. f: Atom → 𝔹
  2. s: Spec
  3. b: Env → bar(ExpValue(boolean))
```

```
  4. c: Stmt
  5. d: Stmt
  6. en: Env
  ⊢ b en ∈ bar(ExpValue(boolean))
  |
  BY Auto

*A var_assig   (v:t) := e ==
                 λen.case e en
                     of Exc(en',exc) -> StmtValueExc(en';exc)
                      | Make(en',o) -> StmtValueNorm(en'{(v:t)
                -> o})

*T var_assig_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀v:Atom. ∀t:Type. ∀e:Exp(t).
|    (v:t) := e ∈ Stmt
|
BY Unfold `var_assig` 0 THEN Auto

*C stmt_end    **************************************
```

```
*C hoare_begin       ************ HOARE ************
*C This theory defines Hoare logic for J language.
*A cond                       Cond ==  Env → ℙ
*A hoare       {P[e]} st {Q[e]} ==
                 ∀p:Env
                   P[p]
                   ⇒ st p in! StmtValue(f;s)
                   ⇒ ↑IsNorm(st p)
                   ⇒ Q[env(st p)]
*T hoare_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀P,Q:Cond. ∀st:Stmt.
|    {P[e]} st {Q[e]} ∈ ℙ
|
BY Unfold ‘hoare‘ 0 THEN AutoT

*T hoare_skip

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀P,Q:Cond.
|    (∀e:Env. P[e] ⇒ Q[e]) ⇒ {P[e]} skip {Q[e]}
|
BY Unfolds ‘‘hoare skip‘‘ 0 THEN Reduce 0 THEN UnivCD THENA Au
|  to
|
1. f: Atom → 𝔹
2. s: Spec
3. P: Cond
4. Q: Cond
5. ∀e:Env. P[e] ⇒ Q[e]
6. p: Env
7. P[p]
8. StmtValueNorm(p) in! StmtValue(f;s)
9. True
⊢ Q[p]
|
BY BHyp 5 THEN Auto

*T env_seq

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀st1,st2:Stmt. ∀p:Env.
|    st1 p in! StmtValue(f;s)
|    ⇒ ↑IsNorm(st1 p)
|    ⇒ st1; st2 p = st2 env(st1 p)
|
BY UnivCD THENA Auto
|
1. f: Atom → 𝔹
2. s: Spec
3. st1: Stmt
4. st2: Stmt
5. p: Env
6. st1 p in! StmtValue(f;s)
7. ↑IsNorm(st1 p)
⊢ st1; st2 p = st2 env(st1 p)
|
```

```
BY Unfold ‘seq‘ 0 THEN Reduce 0
|
⊢ case st1 p
|  of StmtValueExc(en’,ex) -> st1 p
|   | StmtValueNorm(en’) -> st2 en’
|
|  = st2 env(st1 p)
|
BY InstLemma ‘stmt_value_cases_norm‘ [⌈f⌉;⌈s⌉;⌈st1 p⌉;⌈bar(Stm
   tValue(f;s))⌉;
   ⌈λ2en ex.st1 p⌉;⌈λ2en.st2 en⌉] THEN AutoT

*T induce_seq

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀p:Env. ∀st1,st2:Stmt.
|    st1; st2 p in! StmtValue(f;s) ⇒ st1 p in! StmtValue(f;s)
|
BY UnivCD THENA Auto
|
1. f: Atom → 𝔹
2. s: Spec
3. p: Env
4. st1: Stmt
5. st2: Stmt
6. st1; st2 p in! StmtValue(f;s)
⊢ st1 p in! StmtValue(f;s)
|
BY RepUnfolds ‘‘seq stmt_value_cases‘‘ 6 THEN Reduce 6
|
6. let <en’,r> = (st1 p)
   in
   case r of inl(ex) => st1 p | inr(i) => st2 en’
   in! StmtValue(f;s)
|
BY SpreadInduce ‘en’ ‘r’ ⌈case r of inl(ex) => st1 p | inr(i)
   => st2 en⌉
   ⌈StmtValue(f;s)⌉ THEN Auto

*T is_norm_seq

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀st1,st2:Stmt. ∀p:Env.
|    st1; st2 p in! StmtValue(f;s)
|    ⇒ ↑IsNorm(st1; st2 p)
|    ⇒ ↑IsNorm(st1 p)
|
BY UnivCD THENM FLemma ‘induce_seq‘ [6] THENA Auto
|
1. f: Atom → 𝔹
2. s: Spec
3. st1: Stmt
4. st2: Stmt
5. p: Env
6. st1; st2 p in! StmtValue(f;s)
7. ↑IsNorm(st1; st2 p)
```

```
8. st1 p in! StmtValue(f;s)
⊢ ↑IsNorm(st1 p)
|
BY RepUnfolds ``stmt_value_is_norm seq`` 7 THEN Reduce 7
|
7. ↑case case st1 p
         of StmtValueExc(en',ex) -> st1 p
          | StmtValueNorm(en') -> st2 en'

    of StmtValueExc(en,ex) -> ff
     | StmtValueNorm(en) -> tt

|
BY Decide ⌈↑IsNorm(st1 p)⌉ THEN Auto
|
9. ¬↑IsNorm(st1 p)
|
BY InstLemma `stmt_value_cases_exc` [⌈f⌉;⌈s⌉;⌈st1 p⌉;⌈bar(Stmt
|  Value(f;s))⌉;
|  ⌈λ₂en.st1 p⌉;⌈λ₂en.st2 en⌉] THEN Reduce 0 THENA AutoT
|
10. case st1 p
    of StmtValueExc(en,ex) -> st1 p
     | StmtValueNorm(en) -> st2 en

    = st1 p
|
BY InstLemma `bar_eq_lemma_rev` [⌈StmtValue(f;s)⌉
|  ] THENM FHyp (-1) [10] THENM Thin (-2) THENM Thin 10 THENA
|  Auto
|
10. case st1 p
    of StmtValueExc(en,ex) -> st1 p
     | StmtValueNorm(en) -> st2 en

    = st1 p
|
BY HypSubst 10 7 THENA Auto
|
7. ↑case st1 p
    of StmtValueExc(en,ex) -> ff
     | StmtValueNorm(en) -> tt

|
BY Fold `stmt_value_is_norm` 7 THEN Auto

*T hoare_seq

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀P,Q:Cond. ∀st1,st2:Stmt.
|    (∃R:Cond. {P[en]} st1 {R en} ∧ {R en} st2 {Q[en]})
|    ⇒ {P[en]} st1; st2 {Q[en]}
|
BY Unfold `hoare` 0 THEN UnivCD THENM D 7 THENM D 8 THENA Auto
|  T
|
```

```
1. f: Atom → 𝔹
2. s: Spec
3. P: Cond
4. Q: Cond
5. st1: Stmt
6. st2: Stmt
7. R: Cond
8. ∀p:Env
     P[p]
     ⇒ st1 p in! StmtValue(f;s)
     ⇒ ↑IsNorm(st1 p)
     ⇒ R env(st1 p)
9. ∀p:Env
     R p
     ⇒ st2 p in! StmtValue(f;s)
     ⇒ ↑IsNorm(st2 p)
     ⇒ Q[env(st2 p)]
10. p: Env
11. P[p]
12. st1; st2 p in! StmtValue(f;s)
13. ↑IsNorm(st1; st2 p)
⊢ Q[env(st1; st2 p)]
|
BY FLemma ‘induce_seq‘ [12] THENM FLemma ‘is_norm_seq‘ [13] TH
|  ENA Auto
|
14. st1 p in! StmtValue(f;s)
15. ↑IsNorm(st1 p)
|
BY InstLemma ‘env_seq‘ [⌈f⌉;⌈s⌉;⌈st1⌉;⌈st2⌉;⌈p⌉] THENA Auto
|
16. st1; st2 p = st2 env(st1 p)
|
BY InstLemma ‘bar_eq_lemma‘ [⌈StmtValue(f;s)⌉
|  ] THENM FHyp 17 [16] THENM Thin (-2) THENM Thin 16 THENA Au
|  to
|
16. st1; st2 p = st2 env(st1 p)
|
BY RWH (HypC 16) 0 THENA Auto
|
⊢ Q[env(st2 env(st1 p))]
|
BY BHyp 9 THENA AutoT
|\
| ⊢ R env(st1 p)
| |
1 BY BHyp 8 THEN Auto
|\
| ⊢ st2 env(st1 p) in! StmtValue(f;s)
| |
1 BY RWH (RevHypC 16) 0 THEN Auto
 \
```

```
  ⊢ ↑IsNorm(st2 env(st1 p))
  |
  BY RevHypSubst 16 0 THEN Auto

*T hoare_var_assig_int

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀P,Q:Cond. ∀v:Atom. ∀z:ℤ.
|    (∀en:Env. P[en] ⇒ Q[en{(v:int) -> JConst(z;int)}])
|    ⇒ {P[en]} (v:int) := (z) {Q[en]}
|
BY RepUnfolds ``hoare var_assig int_const const_exp`` 0 THEN R
|  educe 0 THEN UnivCD T
|  HENA Auto
|
1. f: Atom → 𝔹
2. s: Spec
3. P: Cond
4. Q: Cond
5. v: Atom
6. z: ℤ
7. ∀en:Env. P[en] ⇒ Q[en{(v:int) -> JConst(z;int)}]
8. p: Env
9. P[p]
10. StmtValueNorm(p{(v:int) -> JConst(z;int)})
    in! StmtValue(f;s)
11. True
⊢ Q[p{(v:int) -> JConst(z;int)}]
|
BY BHyp 7 THEN Auto

*T hoare_var_assig_var

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀P,Q:Cond. ∀t:Type. ∀v,w:Atom.
|    (∀en:Env. P[en] ⇒ Q[en{(v:t) -> en w t}])
|    ⇒ {P[en]} (v:t) := (w:t) {Q[en]}
|
BY RepUnfolds ``hoare var_assig var_exp`` 0 THEN Reduce 0 THEN
|   UnivCD THENA Auto
|
1. f: Atom → 𝔹
2. s: Spec
3. P: Cond
4. Q: Cond
5. t: Type
6. v: Atom
7. w: Atom
8. ∀en:Env. P[en] ⇒ Q[en{(v:t) -> en w t}]
9. p: Env
10. P[p]
11. StmtValueNorm(p{(v:t) -> p w t}) in! StmtValue(f;s)
12. True
⊢ Q[p{(v:t) -> p w t}]
|
BY BHyp 8 THEN Auto
```

```
*T and_hoare

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀P1,P2,Q1,Q2:Cond. ∀st:Stmt.
|    {P1[en]} st {Q1[en]}
|    ⇒ {P2[en]} st {Q2[en]}
|    ⇒ {P1[en] ∧ P2[en]} st {Q1[en] ∧ Q2[en]}
|
BY Unfold ‘hoare‘ 0 THEN AutoT
|\
| 1. f: Atom → 𝔹
| 2. s: Spec
| 3. P1: Cond
| 4. P2: Cond
| 5. Q1: Cond
| 6. Q2: Cond
| 7. st: Stmt
| 8. ∀p:Env
|      P1[p]
|      ⇒ st p in! StmtValue(f;s)
|      ⇒ ↑IsNorm(st p)
|      ⇒ Q1[env(st p)]
| 9. ∀p:Env
|      P2[p]
|      ⇒ st p in! StmtValue(f;s)
|      ⇒ ↑IsNorm(st p)
|      ⇒ Q2[env(st p)]
| 10. p: Env
| 11. P1[p]
| 12. P2[p]
| 13. st p in! StmtValue(f;s)
| 14. ↑IsNorm(st p)
| ⊢ Q1[env(st p)]
| |
1 BY FHyp 8 [11;14] THEN Auto
 \
  1. f: Atom → 𝔹
  2. s: Spec
  3. P1: Cond
  4. P2: Cond
  5. Q1: Cond
  6. Q2: Cond
  7. st: Stmt
  8. ∀p:Env
       P1[p]
       ⇒ st p in! StmtValue(f;s)
       ⇒ ↑IsNorm(st p)
       ⇒ Q1[env(st p)]
  9. ∀p:Env
       P2[p]
       ⇒ st p in! StmtValue(f;s)
       ⇒ ↑IsNorm(st p)
       ⇒ Q2[env(st p)]
  10. p: Env
  11. P1[p]
```

```
  12. P2[p]
  13. st p in! StmtValue(f;s)
  14. ↑IsNorm(st p)
  ⊢ Q2[env(st p)]
  |
  BY FHyp 9 [13;14] THEN Auto

*C hoare_end                  *******************************
```

```
*C verify_1_begin ********* VERIFY_1 *********
*C This theory gives several very simple verification
   examples using Hoare logic.
*T example_1

⊢ ∀f:Atom → 𝔹. ∀s:Spec.
|    {True} ("x":int) := (27) {en "x" int = JConst(27;int)}
|
BY UnivCD THEN Hoare THEN Auto

*T example_2

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀o:J(int).
|    {en "y" int = o} ("x":int) := (27) {en "y" int = o}
|
BY UnivCD THENM Hoare THEN Auto

*T example_3

⊢ ∀f:Atom → 𝔹. ∀s:Spec.
|    {True} ("y":int) := (27); ("x":int) := (27) {en "x" int
|    = en "y" int}
|
BY UnivCD THENM HoareSeq ⌈λen.en "y" int = JConst(27;int)⌉ THE
   NM Hoare THEN Auto

*T example_4

⊢ ∀f:Atom → 𝔹. ∀s:Spec.
|    {True} ("y":int) := ("x":int); ("z":int) := ("y":int) {en
|    "x"
|    int
|    = en "z" int}
|
BY UnivCD THENM HoareSeq ⌈λen.en "y" int = en "x" int⌉ THENM H
   oare THEN Auto

*C verify_1_end
               **********************************
```

```
*C rec_pair_begin         ************ REC_PAIR ************
*C More verification examples.
*D rec_pair_names_df   rec_pair_names()== rec_pair_names{}
*A rec_pair_names       rec_pair_names() == λa.a =a "RecPair"
*T rec_pair_names_wf

⊢ rec_pair_names() ∈ Atom → 𝔹
|
BY Unfold ‘rec_pair_names‘ 0 THEN Auto

*D rec_pair_spec_df           rec_pair_spec()== rec_pair_spec{}
*A rec_pair_spec
               rec_pair_spec() ==
                 λn,a.
                  if a =a "z" then inl int
                  if a =a "r" then inl Class("RecPair")
                  else inr ·
                  fi
*T rec_pair_spec_wf

⊢ rec_pair_spec() ∈ Spec
|
BY Unfold ‘rec_pair_spec‘ 0 THEN Auto
|
1. n: JClassName(rec_pair_names())
2. a: Atom
⊢ "RecPair" ∈ JClassName(rec_pair_names())
|
BY Unfold ‘j_class_name‘ 0 THEN MemTypeCD THENA Auto
|\
| ⊢ "RecPair" ∈ Atom
| |
1 BY Auto
 \
  ⊢ ↑(rec_pair_names() "RecPair")
  |
  BY Unfold ‘rec_pair_names‘ 0 THEN Reduce 0
  |
  ⊢ ↑"RecPair" =a "RecPair"
  |
  BY RWH bool_to_propC 0 THEN Auto

*D rec_pair_df                RecPair== rec_pair{}
*A rec_pair                   RecPair ==  Class("RecPair")
*T rec_pair_wf

⊢ RecPair ∈ Type
|
BY Unfold ‘rec_pair‘ 0 THEN Auto
|
⊢ "RecPair" ∈ JClassName(rec_pair_names())
|
BY MemTypeCD THEN Auto
|
⊢ ↑(rec_pair_names() "RecPair")
```

```
|
BY Unfold `rec_pair_names` 0 THEN Reduce 0 THENM
    RWH bool_to_propC 0 THEN Auto

*D new_df                    new(<t:t:*>)== new{}(<t>)
*A new         new(t) ==  λen.ExpValueMake(λa,s.
                 [cor(JConst(0;int)):=Jac(0)@en a s];t0)
*T new_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀t:Type.  new(t) ∈ Exp(t)
|
BY Unfold `new` 0 THEN Auto
|
1. f: Atom → 𝔹
2. s: Spec
3. t: Type
4. en: Env
5. a: Atom
6. s1: Type
⊢ [cor(JConst(0;int)):=Jac(0)@en a s1] ∈ J(s1)
|
BY InstLemma `j_c_update_wf` [⌈f⌉;⌈s⌉;⌈int⌉;⌈s1⌉;⌈JConst(0;int)⌉;
   ⌈Jac(0)⌉;⌈en a s1⌉
|  ] THEN Auto
|
⊢ Jac(0) ∈ JCor(int)
|
BY InstLemma `j_ac_wf` [⌈f⌉;⌈int⌉;⌈0⌉] THEN Auto

*D if_exp_df   if_exp(<b:b:*>;<c:c:*>;<d:d:*>;<f:f:*>;<s:s:*>)
               == if_exp{}(<b>; <c>; <d>; <f>; <s>)
*A if_exp      if_exp(b;c;d;f;s) ==
                 λen.case b en
                     of Exc(en',ex) -> ExpValueExc(en';ex)
                      | Make(en',bv) ->
                     case get_j_cor(bv;f;s)
                     of Jc -> ExpValueExc(en';
                                  JConst("NilPointerException";
               Exception))
                      | Jac(bav) -> if bav then c en' else d en' fi

*T if_exp_wf

⊢ ∀f:Atom → 𝔹. ∀s:Spec. ∀t:Type.
 ∀b:Env → ExpValue(boolean). ∀c,d:Env → ExpValue(t).
|    if_exp(b;c;d;f;s) ∈ Env → ExpValue(t)
|
BY Unfolds ``exp if_exp`` 0 THEN UnivCD THENM MemCD
   THENM MemCD THEN Auto
|
1. f: Atom → 𝔹
2. s: Spec
3. t: Type
4. b: Env → ExpValue(boolean)
5. c: Env → ExpValue(t)
```

```
6. d: Env → ExpValue(t)
7. en: Env
8. en': Env
9. bv: J(boolean)
⊢ case get_j_cor(bv;f;s)
|  of Jc -> ExpValueExc(en';
            JConst("NilPointerException";Exception))
|   | Jac(bav) -> if bav then c en' else d en' fi
|   ∈ ExpValue(t)
|
BY InstLemma `j_cor_cases_wf` [⌈f⌉;⌈boolean⌉;⌈ExpValue(t)⌉;
|  ⌈ExpValueExc(en';JConst("NilPointerException";Exception))⌉;
 ⌈λ2bav.if bav
|                                    then c en'
|                                    else d en'
|                                    fi ⌉;
|  ⌈get_j_cor(bv;f;s)⌉] THEN Auto
|
⊢ JActCor(boolean) ∈ 𝕌{18}
|
BY Reduce 0 THEN Auto

*D sigma_df    sigma(<p:p:*>;<n:n:*>)== sigma{}(<p>; <n>)
*M sigma_ml
               sigma(p;n)
               ==r if_exp(n == (0);(0);((p."z") +
               sigma((p."r");(n - (1))));rec_pair_names();re
               c_pair_spec())
*T sigma_wf

⊢ ∀n:ℕ. ∀p:Env → ExpValue(RecPair).  sigma(p;(n)) ∈
 Env → ExpValue(int)
 Proof is suppressed, but it's available on Web.

*C rec_pair_end         **********************************
```

Bibliography

[1] ACM. *The Second ACM History of Programming Languages Conference*, 1996.

[2] Appel and Haken. Every planar map is four colorable. *Contemporary Mathematics*, 98, 1989.

[3] R. Boyer and J. Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[4] R. Cohen. The defensive java virtual machine specification. Technical report, Computational Logic Inc., 1997.

[5] P. Constable, P. Jackson, P. Naumov, and J. Uribe. Constructively formalizing automata. In *Proof Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1997.

[6] R.L. Constable and K. Crary. Computational complexity and induction for partial computable functions in type theory. In *Symposium on Logic in Computer Science*, 1998. (Submitted).

[7] T. Coquand and C. Paulin. Inductively defined types. In P. Martin-Lof and G. Mints, editors, *COLOG-88*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66, December 1988.

[8] R.L. Crole. Recursive types via fixpoint objects. In *CLICS Workshop*. Aarhus University, Denmark, May 1992.

[9] R.L. Constable et al. *Implementing Mathematics with Nuprl Proof Development System*. Printice Hall, 1986.

[10] M. Gordon and T. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[11] J. Gosting, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[12] J. Greiner. Programming with inductive and co-inductive types. Technical report, School of Computer Science, Carnegie Mellon University, January 1992. CMU-CS-92-109.

[13] J. Guttman and M. Wand, editors. *VLISP: A Verified Implementation of Scheme*. Kluwer Academic Publishers, 1995.

[14] T. Hagino. A typed lambda calculus with categorical type constructor. In A. Poinge D.H. Pitt, editor, *Category Theory and Computer Science*, volume 283 of *Lecture Notes in Computer Science*, pages 140–157, September 1987.

[15] C.A.R. Hoare. An axiomatic basis for computer programming. In *Comm. ACM*, pages 576–583. ACM, 1969.

[16] Isabelle online theory library. http://www4.informatik.tu-muenchen.de / isabelle.

[17] *Journal of Formalized Mathematics*. Warsaw University, Bialystok, Poland, 1989 -.

[18] N. Klarlund and M. Schwartzbach. Graph types. In *20th ACM Symp. on Princ. of Programming Languages*, pages 196–205, 1993.

[19] D. Knuth. *The TEXbook*. Addison-Wesley, 1984.

[20] G Kolata. With major math proof, brute computers show flash of reasoning power. *New York Times*, pages C–1, December 10 1996.

[21] L. Lamport. *LATEX: a document preparation system*. Addison-Wesley, 1986.

[22] G.W. Leibniz. *Logical Papers: A Selection*. Clarendon Press, Oxford, 1966. Edited and translated by G. Parkinson.

[23] G.W. Leibniz. *Mathematischer, naturwissenschaftlicher und technischer Briefwechsel*, volume 1, chapter Accessio ad arithmeticam infinitorum, pages 1–19. Akademie Verlag, Berlin, 1976.

[24] W. McCune. Solution of the Robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

[25] E. Mendelson. *Introduction to Mathematical Logic*. Van Nostrand, 1964.

[26] P.F. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, 1988.

[27] L. Mikušiak and M. Adámy. Publishing formal specifications in z notation on world wide web. In M. Bidoit and M. Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development*, pages 871–874. Springer, 1997.

[28] P. Naumov. Simple imperative programming language. http://www.cs.cornell.edu /Info/Projects/NuPrl/Nuprl4.2/Libraries/Semantics.

[29] P. Naumov. Turing theories. http://www.cs.cornell.edu /Info/Projects/NuPrl/Nuprl4.2/Libraries/Turing.

[30] T. Nipkow and D. Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages p. 161–170. ACM Press, New York, 1998.

[31] D. Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998. To appear.

[32] J. M. Rushby S. Owre and N. Shankar. Pvs: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Springer Verlag Lecture Notes in Artificial Intelligence*, pages 748–752, 1992.

[33] S. Smith. *Partial Objects in Type Theory*. PhD thesis, Cornell University, 1988.

[34] M. Srivas and S. Miller. Applying formal verification to a commercial microprocessor. In *IFIP International Conference on Computer Hardware Description Languages*, pages 493–502, Chiba, Japan, August 1995.

[35] R. Stärk. A logic program theorem prover. http://www2-iiuf.unifr.ch /tcs/robert.staerk/lptp/index.html.

[36] D. Syme. Proving java type soundness. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998. To appear.

[37] The CtCoq User Interface. http://www.inria.fr /croap/ctcoq/ctcoq-eng.html.

[38] The QED project. http://www.mcs.anl.gov/qed.

[39] R. Vaughn and D. Svitavsky. Nuprl/www browser design. http://www.cs.cornell.edu /Info/Projects/NuPrl/html/design.html.

[40] Z browser plug-in. http://www.ora.on.ca /z-eves/zbplugin.html.