INSTITUT FÜR INFORMATIK

UND PRAKTISCHE MATHEMATIK

Programmiersprachen und Rechenkonzepte

Bad Honnef, 3.-5. Mai 2004

Wolfgang Goerigk (Hrsg.)

Bericht Nr. 0410
Januar 2005

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

KIEL

Institut für Informatik und Praktische Mathematik der
Christian-Albrechts-Universität zu Kiel
Olshausenstraße 40, D-24098 Kiel
Germany

Programmiersprachen und Rechenkonzepte

Physikzentrum Bad Honnef

3. – 5. Mai 2004

Wolfgang Goerigk (Hrsg.)

wg@informatik.uni-kiel.de

Bericht Nr. 0410
Januar 2005

Dieser Bericht enthält eine Zusammenstellung der Beiträge des Workshops *Programmiersprachen und Rechenkonzepte* im Mai 2004. Der Bericht ist als persönliche Mitteilung aufzufassen.

Vorwort

Die GI-Fachgruppe 2.1.4 *Programmiersprachen und Rechenkonzepte* veranstaltete vom 3. bis 5. Mai 2004 im Physikzentrum Bad Honnef ihren jährlichen Workshop. Dieser Bericht enthält eine Zusammenstellung der Beiträge. Das Treffen diente wie in jedem Jahr gegenseitigem Kennenlernen, der Vertiefung gegenseitiger Kontakte, der Vorstellung neuer Arbeiten und Ergebnisse und vor allem der intensiven Diskussion.

Ein breites Spektrum von Beiträgen, von theoretischen Grundlagen über Programmentwicklung, Sprachdesign, Softwaretechnik und Objektorientierung bis hin zur überraschend langen Geschichte der Rechenautomaten seit der Antike bildete ein interessantes und abwechlungsreiches Programm. Unter anderem waren imperative, funktionale und funktional-logische Sprachen, Software/-Hardware-Codesign, Semantik, Web-Programmierung und Softwaretechnik, generative Programmierung, Aspekte und formale Testunterstützung Thema. Interessante Beiträge zu diesen und weiteren Themen gaben Anlaß zu Erfahrungsaustausch und Fachgesprächen auch mit den Teilnehmern des zeitgleich im Physikzentrum Bad Honnef stattfindenden Workshops „Reengineering“.

Allen Teilnehmern möchte ich dafür danken, daß sie mit ihren Vorträgen und konstruktiven Diskussionsbeiträgen zum Gelingen des Workshops beigetragen haben. Dank für die Vielfalt und Qualität der Beiträge gebührt den Autoren. Ein Wort des Dankes gebührt ebenso den Mitarbeitern und der Leitung des Physikzentrums Bad Honnef für die gewohnte angenehme und anregende Atmosphäre und umfassende Betreuung.

Kiel im Januar 2005 Wolfgang Goerigk

Inhaltsverzeichnis

Ein Uberblick über die Entwicklung der Rechenautomaten

Wolfram-M. Lippe
Institut für Informatik, Universität Münster
lippe@uni-muenster.de

1. Zur Geschichte des Begriffes „Informatik“

Der Gedanke, einen Beitrag zur Geschichte der Rechenautomaten zu erstellen, hatte viele Väter:

Zum einen war es die seit jeher vorhandene Faszination für alte Techniken. Es ist immer wieder bewundernswert, mit welcher Energie der Mensch mit einfachsten technischen Hilfsmitteln naturwissenschaftliche und technische Höchstleistungen vollbracht hat. Genialität, Akribie und Fleiß waren die bestimmenden Faktoren. Viele Erkenntnisse gingen aber auch im Laufe der Zeit verloren und mussten zum Teil neu entdeckt werden. Selbst in dem noch so jungen und sich mit unheimlicher Geschwindigkeit fortentwickelnden Fach der „ Informatik“ gab es immer wieder weit vorausschauende Konzepte, die „ noch nicht“ verstanden wurden oder wegen des zu diesem Zeitpunkt gegebenen technologischen Umfeldes noch nicht realisierbar waren und daher wieder in Vergessenheit gerieten, um sodann später wieder neu entdeckt und unter neuem Namen erfolgreich zu werden. Diese Entwicklung konnte ich zu einem erheblichen Teil noch selbst mitverfolgen. Meine ersten „ Gehversuche“ als Programmierer erfolgten 1965 auf einer Röhrenmaschine vom Typ Zuse Z22 an der Universität des Saarlandes in Saarbrücken. Die Programmiersprache war ALGOL 60 und das Eingabemedium ein Fernschreiber mit einem 5-Kanal-Lochstreifen. In immer kürzeren Abständen folgten neue Modelle und neue Technologien mit denen ich mich vertraut machen mußte: Electrologica X1, CDC 3300, CDC 6600, Telefunken TR440, PDP 10 und 11, Siemens 6660 usw.

Ein weitere Grund war die Feststellung der Gesellschaft für Informatik in ihrer Festschrift anlässlich ihres 30- jährigen Bestehens, daß ich wohl der erste Student war, der in Deutschland ein Diplom im Fach Informatik abgelegt hat. Dies und die Erfahrungen, die ich durch die Mitwirkung am Aufbau der Informatik-Abteilungen an den Universitäten Saarbrücken, Kiel, Münster und Gießen gesammelt habe, führte dazu, daß ich mich intensiver mit der Geschichte der Informatikentwicklung an deutschen Hochschulen beschäftigte und damit auch mit der Geschichte der Geräte und Techniken,

W. Goerigk (Hrsg.): Programmiersprachen und Rechenkonzepte. Bericht Nr. 0410, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

die die Informatik verwendet, denn diese Geschichte ist wesentlich älter als die eigentliche Informatik.

Das Wort „ Informatik“ selbst war vor 1950 kaum in Gebrauch. Sein erster Gebrauch liegt im Dunkeln; seine Entstehung durch Anhängen der Endung ‘ -ik’ an den Stamm des Wortes „ Information“ scheint aber klar zu sein. Eine frühe Verwendung findet sich durch Karl Steinbuch. Nachdem der Begriff „ Informatik“ gegen Ende der fünfziger Jahre für Erzeugnisse der Firma Standard Elektrik Lorenz (SEL) urheberrechtlich geschützt wurde, war das Wort einer breiten Verwendung in Deutschland entzogen. Mitte der sechziger Jahre wurde im Deutschen das Wort „ Informationsverarbeitung“ mehr und mehr gebräuchlich, in direkter Entsprechung zu ‘ Information Processing’ - ein Wort, das sich auch im Namen eines internationalen Verbandes, der IFIP (International Federation of Information Processing) wiederfindet - sowie parallel hierzu auch der Begriff „ Kybernetik“ , der vor allem in Arbeiten von Steinbuch Verwendung fand.

In Frankreich tat man sich mit dem Pendant „ Traitement de l’ information“ besonders schwer, und man empfand dort allgemein Erleichterung, als die Académie Française das prägnante Wort „ informatique“ einführte:

L’ informatique:
Science de traitement rationel, notamment par machines automatiques, de l’ information considérée comme le support des connaissances humaine et des communications, dans les domains techniques, économiques et sociale.

Es ist nicht bekannt, ob die Académie Française den Begriff „ Informatik“ zum Vorbild hatte, aber durch diese Entscheidung wurde der Begriff „ Informatik“ in Deutschland wiederentdeckt und zunächst vor allem in akademischen Zirkeln schnell hoffähig. Als der damalige Bundesminister für Bildung und Wissenschaft, Gerhard Stoltenberg, 1968 anläßlich der Eröffnung einer Tagung in Berlin das Wort „ Informatik“ mehrfach gebrauchte, wurde es von Journalisten aufgegriffen und war bald auch über die Fachwelt hinaus existent. Es wurde dann auch für den Namen desjenigen Förderprogramms der Bundesregierung verwandt, mit dem ab Mitte der sechziger Jahre versucht wurde, den Rückstand Deutschlands in der Informationstechnologie aufzuholen und mit dem u.a. in größerem Rahmen die Erstausstattung der deutschen Universitäten mit Rechnern finanziert wurde.

In den USA konnte sich die parallele Konstruktion ‘ Informatics’ nicht durchsetzen - auch sie war im übrigen durch Firmennamen besetzt. Statt dessen wurde zunächst der Begriff ‘ Computing Science’ verwendet, der danach durch ‘ Computer Science’ verdrängt wurde. Erst in neuerer Zeit tritt ‘ Informatics’ , z.B. in Form von „ Applied Informatics“ , wieder in den

Vordergrund. In Großbritannien ist dagegen vor allem der Ausdruck „ Information Technology“ verbreitet.

Die Herkunft vieler anderer Begriffe innerhalb der Informatik lässt sich genauer angeben. Wie in unserer heutigen Zeit weit verbreitet, stehen vor allem Abkürzungen englischer Ausdrücke im Vordergrund: ROM (Read Only Memory) für Speicherbausteine, FORTRAN (…..) für die erste höhere Programmiersprache usw. Auch bei berühmten Personen der Geschichte wurden Anleihen gemacht: So wurden Programmiersprachen nach dem Mathematiker und Erbauer einer der ersten digitalen Rechenmaschinen Claude PASCAL bzw. nach ADA, einer Mitarbeiterin von Charles Babbage, der im 19. Jahrhundert als erster das Konzept für einen programmierbaren Rechner entwickelte, benannt

Auch der für die Informatik essentielle Begriff des „ Algorithmus“ besitzt eine interessante Herkunftsgeschichte. Er stammt nicht, wie von vielen auf Grund der Endung – us angenommen, aus dem Lateinischen, sondern aus dem Arabischen. Er geht auf den Namen eines Mathematikers zurück, der zu Zeiten des Kalifen al-Mamun in Bagdad im sog. „ Haus der Weisen“ - heute würden wir dazu Universität sagen - lebte. Sein Name war Ibn Musa Djafar al-Choresmi (auch Al Khawarizmi, al- Khowarizmi, al- Hwarazmi geschrieben), geboren etwa 780, gestorben etwa 850. Er stammte aus dem südöstlichen des Aral-Sees gelegenen Choresmien in der heutigen Republik Usbekistan. In Bagdad schrieb er das Werk „ Aufgabensammlung für Kaufleute und Testamentsvollstrecker“ , welches in manchen Bezeichnungen und in seiner algebraisierenden Tendenz auch den oben erwähnten indischen Einfluß zeigt. Dieses Buch wurde, wie viele andere arabische Lehrbücher auch, gegen Ende des Mittelalters in das Lateinische übersetzt und erhielt den Titel „ liber algorithmi“ .

2. Das Räderwerk von Antikythera – der älteste Rechner der Welt

Um die Jahrhundertwende wurden in der Ägäis nahe der Insel Antikythera (antik: Aegil) von einem römischen Schiffswrack bronzene Teile geborgen, die in keiner Weise mit dem verglichen werden konnten, was bis dahin im Mittelmeerraum gefunden wurde. Das besagte Schiffswrack gab auch eine Reihe anderer Kunstwerke frei, z.B. zahlreiche Statuen und Amphoren. Als man sich jedoch anschickte, die erwähnten Bronzeteile zu analysieren, begann ein großes Staunen. Es handelte sich hierbei um ein Räderwerk, das später „ Das Räderwerk von Antikythera“ (nach dem Fundort) oder „ Planetarium“ (nach den Inschriften) genannt wurde. Die geborgenen Fragmente befinden sich heute im National-Museum in Athen.

Man fand in den griechischen Beschriftungen auf den Überresten Hinweise auf den damals gebräuchlichen Kalender, auf Sonne, Mond und auf die damals bereits bekannten Planeten. Daneben fanden sich kreisförmige Skalen mit der Tierkreisteilung einerseits und dagegen verschiebbar - konzentrisch hierzu - Skalen mit den Monatsnamen. Auf der Rückseite des deswegen auch als Planetarium bezeichneten Räderwerks fanden sich sogar 4 konzentrische gegeneinander verschiebbare Ringe, die dann auf andere Himmelskörper und auf die Planeten hinwiesen.

Das eigentliche Räderwerk basiert auf einer Vielzahl von Zahnrädern in 60 Grad-Verzahnung. Diese Verzahnung hat zwar keinen guten Wirkungsgrad, stellt jedoch angesichts der Tatsache, daß über derartige Zahnradtechniken bei den Griechen keinerlei andere Kunde existiert, eine weitere echte Sensation dar. Zahnradtechniken waren den Griechen zwar bekannt, aber sie wurden nur in relativ simplen Anwendungen benutzt. Sie verwendeten Zahnradpaare, z.B. um Kraft in einem rechten Winkel zu übertragen, wie in einer Wassermühle.

Als sich der namhafte Archäologe Spyridon Stais am 17. Mai 1902 an die Untersuchung der Fragmente machte und kurz darauf seine Ergebnisse veröffentlichte, wurden zunächst von vielen Fachleuten die Ergebnisse und die Datierung angezweifelt. Selbst von Fälschung wurde gesprochen. Dennoch sind die Authentizität und die Datierung inzwischen gesichert. Sowohl die gefundenen Münzen als auch die Beschriftung des Gehäuses lassen das Räderwerk auf ca. 80 v. Chr. ansetzen.

Die Untersuchungen brachten auch die Tatsache zu Tage, daß das Gerät auch tatsächlich in Betrieb war. Man fand z.B. zwei Stellen im Getriebe, die repariert worden waren. So ist etwa ein Zahn ersetzt worden. An anderer Stelle wieder ist offenbar die Speiche eines Zahnrades gebrochen gewesen und schließlich durch sorgfältige Einfügung wieder ersetzt worden.

Der überraschende Fund von Antikythera zeigt, daß es theoretische und technologische Erkenntnisse und Fertigkeiten bereits zur Zeit Christi gab, die man bis zu diesem Fund nicht für möglich gehalten hatte. Inzwischen sind einige weitere Analysen erfolgt, so z.B. von Derek de Solla Price, die beweisen, daß das Räderwerk von Antikythera von extrem komplexer arithmetischer Konzeption war, bei der bekannte astronomische Relationen und insbesondere Perioden mit Hilfe von Zähne-Anzahlen realisiert wurden. Es enthält sogar Reste eines Differentialgetriebes (zur Bildung von Differenzen), wie es erst 1832 in England zum Patent angemeldet wurde.

Hinsichtlich seiner Funktion wurde lange spekuliert. Einige Dinge waren von Beginn an klar. Die einzigartige Wichtigkeit des Objekts war offensichtlich und das Getriebe war eindrucksvoll komplex. Aufgrund der Inschriften und

der Zifferblätter ist der Mechanismus korrekt als ein astronomisches Gerät identifiziert worden. Die erste Mutmaßung war, daß es sich hierbei um eine Art Navigationsinstrument, vielleicht ein Astrolabium handelte. Einige dachten, daß es möglicherweise ein kleines Planetarium sein könne, derart, wie Archimedes eines erstellt haben soll. Eine genaue Untersuchung wurde aber erst 1958 durch den Engländer Derek del Solla Price - heute Professor für Wissenschaftsgeschichte an der amerikanischen Yale University - vorgenommen, der beim Studium der Geschichte wissenschaftlicher Instrumente auf den Mechanismus im Athener Museum gestoßen ist.

Er war sofort von dem Räderwerk begeistert:
"Ein vergleichbares Instrument ist nirgends erhalten", schrieb er, "und ist auch in keinem alten wissenschaftlichen oder literarischen Text erwähnt. Nach allem, was wir über Wissenschaft und Technologie im hellenistischen Zeitalter wissen, dürfte es eine solche Vorrichtung eigentlich nicht geben". Price war so begeistert, daß er über ein Jahrzehnt lang daran arbeitete, die Apparatur anhand der stark beschädigten Bronzefragmente zu rekonstruieren. Doch erst die 1971 von der griechischen Atomenergiekommission angefertigten Röntgenaufnahmen brachten endgültigen Aufschluß über das Zahnradgetriebe.

Fügt man die soweit gesammelten Informationen zusammen, scheint es vernünftig, anzunehmen, daß die Absicht des Antikythera-Mechanismus war, die Berechnung gewisser astronomischer Zyklen zu mechanisieren. Diese Zyklen waren ein starkes Merkmal antiker Astronomie. Diese Zyklen benutzend ist es nun einfach, ein Getriebe zu entwickeln, welches durch ein Zifferblatt gesteuert wird, das einmal jährlich gedreht wird und dabei eine Reihe anderer Zahnräder dreht, welche wiederum Zeiger bewegen, die siderische, synodische und drakonische Monate anzeigen. Tatsache ist, daß diese Art arithmetischer Theorie das zentrale Thema der Astronomie der seleuzidischen Babylonier war, welche den Griechen in den letzten paar Jahrhunderten v.Chr. übermittelt wurde. Solche arithmetischen Schemata sind völlig verschieden von der geometrischen Theorie der Kreise und Epizyklen der Astronomie, welche im wesentlichen griechisch erscheinen. Der Mechanismus ist ähnlich einer bedeutenden astronomischen Uhr oder einem modernen Analogcomputer, der mechanische Teile benutzt, um Berechnungen zu speichern. Es ist wirklich schade, daß man nicht weiß, ob das Gerät automatisch oder per Hand gedreht wurde. Es mag in der Hand gehalten und durch ein Rad an der Seite gedreht worden sein, so daß es wie ein Computer arbeitete, möglicherweise für astrologische Zwecke. Price ist eher der Ansicht, daß das Gerät permanent befestigt war. Vielleicht in einer Statue als Ausstellungsstück. In diesem Falle ist es möglicherweise durch die Kraft einer Wasseruhr oder ähnlichem gedreht worden.

John Glave aus England hat anhand der Rekonstruktion von Price den Versuch unternommen, ein funktionierendes Replika des Original-Mechanismus zu konstruieren. Diese Zahnräder sind nicht wie beim Original aus Bronze, sondern aus Messing gefertigt und sie sind zwischen transparenten Platten angebracht, so daß der Mechanismus sichtbar ist. Inwieweit dieser Versuch einer Rekonstruktion in Details mit dem Original übereinstimmt, läßt sich jedoch nur schwer bewerten.

2. Astrolabien

Nach dem Rechner von Antikythera muß man bis zu dem nächsten bekannten Rechengerät einen großen Zeitsprung bis ca. 700 n. Chr. machen. In Urkunden aus dieser Zeit werden im arabischen Raum zum ersten Mal die sog. Astrolabien erwähnt.

Da durch die Wirren in den Zeiten nach dem Niedergang des Römischen Reiches in Europa nicht nur keine Weiterentwicklung in Wissenschaft und Kultur stattfand, sondern bereits vorhandenes Wissen verloren ging, stammen die wesentlichen Impulse der Mathematik der damaligen Zeit - und dies gilt bis in das späte Mittelalter - aus dem arabischen Raum und wurden von dort nach Europa exportiert. Dies ist auch der Grund, daß wir heute nicht mit römischen, sondern mit arabischen Ziffern rechnen und schreiben. Ferner gelangte auch die Algebra, also das Rechnen mit Buchstaben, aus Arabien nach Europa. Es ist aber anzunehmen, daß die Araber selbst sehr viel von ihren mathematischen Errungenschaften, darunter auch die Algebra, aus dem indischen Raum übernommen haben. Von diesen frühen indischen Hochkulturen und ihren mathematischen und astronomischen Kenntnissen ist aber bis heute noch sehr wenig bekannt.

Aber betrachten wir weiter die Astrolabien. Im Prinzip handelt es sich um einen Analogrechner, der allerdings eine wesentlich geringere Komplexität als das Räderwerk von Antikythera aufweist. Das Astrolabium diente sowohl astronomischen Zwecken als auch zur Navigation. Auf einer Grundplatte befindet sich eine Eingravierung der stereographischen Projektion der Erde mit ihren Längen- und Breitengraden (erste Ansätze zu einer Kartographie, die auf Längen- und Breitengraden beruht, gehen auf Ptolemäus zurück; danach sind sie in Europa erst wieder ab 1400 allgemein gebräuchlich). Darüber drehbar ist ein Gitter angeordnet, das den Fixsternhimmel und die Position bekannter Sterne in Form von Zeigern verkörpert. Die Position der Sonne ist gegeben durch ihren Standort in dem Ekliptik-Kreis, der ebenfalls in das Gitter eingebettet ist und die Tierkreiszeichen neben einer 360-Grad-Teilung trägt. Die Einsatzmöglichkeiten von Astrolabien sind vielfältig: Je nachdem welche Größen bekannt sind, lassen sich die wahre Ortszeit, die Zeit des Auf- bzw. Untergangs der Sonne oder bekannter Gestirne sowie die eigene

Position auf der Erde bestimmen.

Die Astrolabien waren bis Ende des letzen Jahrhunderts in der Schiffahrt im Indischen Ozean im Einsatz. Auch in Europa wurden sie für navigatorische Zwecke sowie für astrologische Bestimmungen häufig eingesetzt. Es gibt verschieden Typen von Astrolabien. Der bei weitem populärste Typ ist wohl das planisphärische Astrolabium, bei dem die Himmelssphäre auf die Ebene des Aquators projiziert wird.

Ein Astrolabium zeigt – korrekt eingestellt -, die Himmelskonfiguration an einem bestimmten Ort zu einer bestimmten Zeit an. Hierzu ist die Himmelskonfiguration auf die Oberfläche des Astrolabiums projiziert, so daß durch Markierungen verschiedene Positionen am Himmel leicht zu finden sind. Um ein Astrolabium zu benutzen, justiert man die beweglichen Teile an ein bestimmtes Datum und eine bestimmte Zeit. Einmal eingestellt, ist der ganze Himmel, der sichtbare und der nicht sichtbare Teil, auf der Oberfläche des Instrumentes zu erkennen und die einzelnen Positionen mit Hilfe von Markierungen leicht zu bestimmen. Dies erlaubt eine große Anzahl astronomischer Probleme in einer visuellen Art zu lösen. Typische Anwendungen eines Astrolabiums beinhalten das Bestimmen der Zeitspanne zwischen Tag und Nacht, das Bestimmen des Zeitpunktes eines Himmelsereignisses wie z.B. Sonnenauf- oder Sonnenuntergang, und als handliches Nachschlagewerk für Himmelspositionen. In den islamischen Ländern wurden Astrolabien auch benutzt, um die Zeiten für die täglichen Gebete und die Richtung nach Mekka zu bestimmen

Die Ursprünge der Astrolabien liegen vermutlich in Griechenland. Apollonius (ca. 225 v.Chr.), der sich intensiv mit Kegelschitten beschäftigte, studierte wahrscheinlich die zur Erstellung von Astrolabiem notwendigen Projektionen. Wesentliche Erkenntnisse gelang auch Hipparchus, der in Nicaea (Heute Iznik in der Türkei) um 180 v.Chr. geboren wurde, aber auf Rhodos studierte und arbeitete. Hipparchus charakterisierte die Projektion als eine Methode um komplexe astronomische Probleme ohne sphärische Trigonometrie zu lösen, und er bewies wahrscheinlich ihre Hauptcharakteristica. Hipparchus hat zwar nicht das Astrolabium erfunden, wohl aber die Projektionstheorie verfeinert. Das älteste Beweisstück für die konkrete Benutzung der stereographischen Projektion (siehe Anhang Astrolabium) ist ein Schriftstück des römischen Autors und Architekten Vitruvius (ca. 88 - ca. 26 v.Chr.). Er beschreibt in *De architectura* eine Uhr, die von Ctesibius in Alexandria hergestellt wurde, und in der eine stereographische Projektion benutzt wurde. Ausführlichere Informationen findet man bei Claudius Ptolemy (ca. 150 n.Chr.). Er schrieb umfassend über Projektionen, in seiner als *Planisphaerium* bekannten Arbeit. In ihr gibt es konkrete Hinweise, daß er ein Astrolabien-ähnliches Instrument besessen haben könnte. Ptolemy verfeinerte außerdem noch die

Fundamentalgeometrie des bis dahin bekannten Erde-Sonne Systems, und schuf damit Grundlagen zur Weiterentwicklung von Astrolabien.

3. Astronomische Uhren und Kirchenrechner

In Europa setzt die Weiterentwicklung, was Rechenanlagen und Automaten betrifft, wesentlich später als im arabischen Raum ein, so ab dem 13. Jahrhundert. Hier zunächst geprägt durch die Entwicklung von Kirchenuhren.

Eines der ältesten Zeitmessgeräte ist die Sonnenuhr: An ihrem Stab wirft die Sonne einen Schatten. Lage und Länge des Schattens zeigen die Position der Sonne in Bezug auf die Erde an – und damit die Zeit. Eine Uhr mit linearer Anzeige ist die Wasseruhr. Bereits die alten Ägypter kannten ein- und auslaufendes Wasser als Maß für die Zeit. In den Klöstern des Mittelalters zeugte eine abbrennende Kerze vom Vergehen der Zeit.

Durch die Erfindung der mechanischen Uhr vollzog sich gegen Ende des 13. Jahrhunderts eine technische Revolution. Die ersten Uhren waren Räderuhren mit Gewichtsantrieb, bei denen als Hemmung eine Spindel diente, die mit zwei Ansätzen in das Steigrad eingriff. Da diese Uhren große Abmessungen besaßen, versahen vor allem die Städte einen ihrer Profan- oder Sakralbauten mit einer derartigen Monumentaluhr. Die Federzuguhr tauchte erstmals in der zweiten Hälfte des 14. Jahrhunderts auf. Die ersten tragbaren Federuhren baute der Nürnberger Schlosser Peter Henlein um 1510; sie waren eiförmig (Nürnberger Ei). Damit war in Europa erstmalig wieder ein technologischer Stand erreicht, der schon ca. 1.500 Jahre früher in Kleinasien erreicht worden war. Dennoch war über weitere Jahrhunderte hinweg auch Sanduhren immer noch im Gebrauch.

Um ihr Prestige zu steigern, erweiterten die Städte ihre Kirchenuhren um zusätzliche technische Neuerungen, um ihnen so einen spektakulären Aspekt zu verleihen. Aus den Kirchenuhren wurden astronomische Uhren. Straßburg gehörte durch den zwischen 1352 und 1354 erfolgten Bau der sogenannten Drei-Königsuhr zu den ersten Städten, die das Exempel einer solchen Errungenschaft abgaben. Die Legende behauptet, daß dem Uhrmacher der astronomischen Uhr nach der Vollendung seines Werkes auf Befehl der hohen Beamtenschaft der Stadt, die danach trachtete, ihn zu hindern, andernorts ein ebensolches Meisterwerk zu schaffen, die Augen ausgestochen worden seien. Ähnlich lautende Geschichten existieren auch für andere astronomische Uhren, wie z.B. Olmütz (ca. 1422), Danzig (ca. 1470), Münster (1542), Lübeck (1566) oder Lyon (1598). Wenn auch diese Legenden kein Fünkchen Wahrheit enthalten, so offenbart sie doch den Stolz der Straßburger auf den Besitz eines Werkes, das in der damaligen Zeit zu

den großen Wundern zählte.

Die astronomischen Uhren erfüllten in der damaligen Zeit für das kirchliche und öffentliche Leben vielseitige Zwecke. Es konnten Jahr, Monat, Tag, Wochentag und Mondphasen abgelesen sowie die Tagesheiligen ermittelt werden. Der auf der Uhr dargestellte Horizont ermöglichte es, die Auf- und Untergangszeiten für Sonne, Mond, Planeten und Fixsterne zu bestimmen. Damit lieferten sie die Grunddaten für astrologische Berechnungen und Prophezeiungen, wie sie damals weit verbreitet waren und durch die sich viele Menschen in ihrem täglichen Tun beeinflussen ließen. Man muß sich vor Augen halten, daß damals niemand über eine eigene Uhr oder einen eigenen Kalender verfügte. Somit bestimmte der Blick auf die weit sichtbare Turmuhr bzw. ihr viertelstündiger Klang den täglichen Rythmus. Der Kalender vermittelte Kenntnisse über den Ablauf des Kirchenjahres mit seinen Feiertagen.

Wie bereits erwähnt, war Straßburg eine der ersten Städte, die ihr Münster mit einer Monumentaluhr versahen. Im Verlauf der darauffolgenden Jahrhunderte haben danach drei astronomische Uhren zum Ruhme der Stadt Straßburg beigetragen. Einen Höhepunkt in der Entwicklung von astronomischen Kirchenuhren stellt hierbei sicherlich die dritte Uhr dar, die einmalig in der Welt über einen besonderen "Kirchenrechner" verfügte, um die beweglichen Kirchenfeiertage des jeweiligen Jahres zu berechnen.

Eine weitere Attraktion war ein krähender flügelschlagender Hahn, der die Bewegungen eines Hahns so gut wiedergab, daß die Perfektion selbst heute Bewunderung hervorruft. Dieser Hahn - vermutlich der älteste noch vollständig erhaltende Automat - ist jetzt im Straßburger Kunstgewerbemuseum zu sehen. Er wurde von Dasypodius auch für die zweite Uhr wieder verwendet. Dieser Hahn war so berühmt, daß er bei anderen Uhren, z.B. in Bern, München, Heilbronn, Lyon oder Prag, nachgeahmt wurde

Als die 2. Uhr wegen Abnutzungserscheinungen stehen blieb wurde Schwilgue als Feinmechanikeringenieur im - für die damalige Zeit bereits stolzen - Alter von einundsechzig Jahren mit der Renovierung der Uhr beauftragt wurde, die er von 1838 bis 1842 vornahm.

Fast unvorstellbar ist die Präzision der Uhr. Die zeitliche Abweichung im Jahr beträgt ungefähr 30 Sekunden. Schwilgués Uhr war ferner die erste der Welt, die de facto alle astronomischen Phänomene berücksichtigte. Dies gilt insbesondere für die komplizierten Bewegungen des Mondes und der Sonne, wobei besonders die Darstellung der scheinbaren oder wahren Bewegung des Mondes komplizierte Berechnungen erforderte, die Schwilgue mechanisch realisieren musste. Die Mondbahn bildet mit der Ekliptik (scheinbaren Sonnenbahn) einen Winkel von 5 Grad, und die Ekliptik einen

Winkel von ca. 23 Grad mit dem Himmelsäquator. Zusätzlich ist die Mondbahn einer Präzessionsbewegung – bezogen auf die Ekliptik – unterworfen und unterliegt noch zusätzlich zahlreichen Anomalien. Daher finden sich in der Uhr – neben dem besonders beschriebenen Kirchenrechner zur Berechnung der beweglichen Feiertage – zahlreiche mechanische Spezialrechner, die spezielle Berechnungen durchführen unter anderem zur Berechnung dieser Anomalien. Die einzelnen Anomalien lassen sich durch sinusoidale Gleichungen beschreiben. Insgesamt gibt es zwei Sonnengleichungen, fünf Mondgleichungen und eine Mondknotenliniengleichung. Der Rechner zur Berechnung dieser Gleichungen ist im Erdgeschoß der Uhr in einer Vitrine untergebracht und trägt die Aufschrift: „ Equations solaires et lunaires“ .

Eine Besonderheit, die die astronomische Uhr des Straßburger Münsters in der Welt einmalig macht, ist der bereits erwähnte und sich links im Sockel befindliche Kirchenrechner (comput ecclésiastique). Er wird von der Uhr nur einmal jedes Jahr und zwar in der Silvesternacht gestartet. Durch ihn werden die beweglichen Kirchenfeiertage des nun folgendes Jahres berechnet und auf dem automatischen Kalender angezeigt. Danach verweilt der "Comput ecclésiastique" wieder in Ruhestellung bis zum nächsten Silvesterabend. Die Einstellung der beweglichen Kirchenfeiertage, insbesondere von Ostern, stellte ein besonderes Problem dar und mußte jährlich bei jeder astronomischen Uhr vorgenommen werden.

Bemerkenswert an dieser Uhr und dem Rechner und sind die Genauigkeit mit der sie konstruiert und gebaut wurden. Daß der Kirchenrechner von Schwilgué in der Tat für die "Ewigkeit" ausgelegt war, zeigen in heutiger Zeit erfolgte Untersuchungen, durch die ersichtlich ist, daß es Komponenten gibt, die erstmalig im Jahre 15200 bewegt werden, um eine dann notwendige Korrektur vorzunehmen. Ein „ Jahr 2000- Problem“ , welches weltweit zu Angstzuständen bei Informatikern und Unternehmen geführt hatte, gab es für Schwilgué nicht.

4. Die ersten digitalen Rechenmaschinen

Einfache digitale Rechengeräte, also Maschinen zur Durchführung einfachster numerischer Berechnungen, existieren unter unterschiedlichen Begriffen und Formen bereits seit über 2000 Jahren in Asien, Rußland, Arabien und dem Mittelmeerraum. Am bekanntesten sind sie unter dem Begriff „ Abakus“ . Der Ursprung des Abakus liegt im dunkeln; man vermutet, daß er im indo-

chinesischen Raum entstand. Im Laufe der Zeit entwickelten sich unterschiedliche Ausprägungen des Abakus in verschiedenen Gebieten. In abgelegenen Basaren ist er selbst heute noch im Einsatz.

Der Abakus ist ein, technologisch gesehen, äußerst einfaches Gerät, bei dem praktisch keinerlei Automatismen realisiert sind. Insbesondere muß der Zehnerübertrag vom Benutzer händisch durchgeführt werden. Erst im 17. Jahrhundert setzte eine Entwicklung ein, die zu richtigen Rechenmaschinen führten, die zur automatischen Durchführung der vier Grundrechenarten in der Lage waren. Gleichzeitig wurde hierdurch die Entwicklung von Tischrechenmaschinen eingeleitet. Zu nennen sind vor allem

Schickard (1592-1635)
Pascal (1623-1662)
Leibniz (1646-1716)

die ihre Maschinen zum Teil unabhängig voneinander entwickelten.

Schickard war mit dem berühmten Astronomen Kepler befreundet und wußte, welche Zeit Kepler in nächtelangen Berechnungen endloser Zahlenkolonnen investierte. Daher konstruierte er um 1623 für ihn eine sechsstellige Addier- und Subtrahiermaschine, die J. Kepler dann bei seinen astronomischen Berechnungen einsetzte. Leider wurde die Maschine kurze Zeit nach ihrer Fertigstellung durch ein Feuer zerstört. Ein zuvor von ihm gebauter Prototyp ging in den Wirren des 30jährigen Krieges verloren.

Die Wiederentdeckung ist dem verstorbenen Keplerforscher Dr. Franz Hammer zu verdanken. Im Jahre 1957 hielt er im Rahmen eines kleinen Kongresses zur Geschichte der Mathematik im Mathematischen Forschungsinstitut Oberwolfach im Schwarzwald einen Vortrag, der alles in Gang brachte.

Hammer berichtete über Unterlagen, die er zumeist schon vor dem Kriege gefunden, aber nicht ausgewertet hatte, aus denen hervorging, daß nicht der große Franzose Blaise Pascal 1642 die erste Rechenmaschine im modernen Sinne dieses Wortes gebaut hat, vielmehr in dessen Geburtsjahr 1623 bereits ein Tübinger Professor, Wilhelm Schickard solches leistete. Hammer legte diese spärlichen Unterlagen dem Kongress vor und schloß mit der Bemerkung, wie die Maschine, von der eine kleine Federskizze, lange verlorene Anlage zu einem Brief Schickard's an Kepler, ein äußerliches Bild gab, im Inneren konstruiert gewesen sei, und ob sie überhaupt funktioniert habe, das werde man wohl niemals erfahren.

Zwei Tage später widerfuhr Bruno Baron v. Freytag Löringhoff, einem der Teilnehmer dieses Kongresses, daß ihm früh am Morgen nach einer

weinseligen Nacht bei erneuter Betrachtung dieser Quellen in wenigen Sekunden alles klar wurde. Der Kongreßleiter Prof. *J. E. Hofmann*, der Mathematikhistoriker und bekannte Bearbeiter des Leibniz-Nachlasses, gab v. Freytag Gelegenheit, noch in den letzten Stunden des Kongresses seinen Rekonstruktionsvorschlag unter allgemeiner Zustimmung vorzutragen.

Selbstverständlich entstand nun der Wunsch, eine Rekonstruktion herzustellen und zu erproben. Das war leichter gesagt als getan und wäre ohne viel Hilfe von mancherlei Seite nie zustande gekommen. Kleine Mißgeschicke hielten die Fertigstellung auf, und so wurde es Januar 1960, bis das erste Exemplar im Auditorium-maximum der Tübinger Universität endlich einem großen Publikum vorgeführt werden konnte.
Eine ähnliche Motivation wie bei Schikard, der seinem Freund Keppler helfen wollte, lag bei Claude Pascal vor, dessen eigentliches Interesse der Mathematik galt. Sein Vater war Steuereintreiber in Paris. Im Gegensatz zu heute bezogen die Steuereintreiber der damaligen Zeit kein festes Gehalt, sondern waren prozentual an den erzielten Steuereinnahmen beteiligt. Da die Steuergesetzgebung schon damals recht kompliziert war, erforderten die einzelnen Berechnungen relativ lange Zeit. Um den Durchsatz und damit das Einkommen seines Vaters zu erhöhen, entwickelte Pascal 1645 eine Rechenmaschine, die ähnlich funktionierte, wie die Maschine von Schickard.

Pascal ließ seine Rechenmaschine in 50 Exemplaren bauen, von denen heute noch neun existieren. Er verbesserte seine nach ihm benannte "Pascaline" ständig, sodaß über Jahrzehnte hinweg fünf- bis zwölfstellige Rechenmaschinen entstanden. Die ersten Pascalinen schenkte er in der Hoffnung auf größere Bekanntheit und Unterstützung bedeutenden Persönlichkeiten, allen voran dem französischen Kanzler sowie der Königin Christine von Schweden. Pascal, der sich zeitweilig in Kreisen des französischen Hofes bewegte, entwickelte aus der Mode des Glücksspiels heraus auch die Grundzüge der Wahrscheinlichkeitsrechnung.

Eine weitere Verbesserung der digitalen Rechenmaschine erfolgte durch Leibniz. Durch die Einführung von Staffelwalzen und beweglichen Schlitten gelang ihm zwischen 1671 (erste Entwürfe) und 1690 (Fertigstellung) der Bau der ersten Maschine für alle vier Grundrechenarten (Vierspeziesmaschine). Leibniz war im übrigen auch einer der ersten, die sich intensiv mit der dualen Darstellung von Zahlen beschäftigte. Weitere digitale Rechenmaschinen wurden von Morland, Grillet, Polini, Leupold, Hahn, Stanhope, Müller und Thomas entwickelt. Versuche im 19. Jahrhundert, ein vorhandenes Original in einen einwandfreien funktionsfähigen Zustand zu versetzen, scheiterten zunächst. Erst im Jahr 1894 konnte man eines der Originale zur einwandfreien Funktion bringen, nachdem die Fertigungstechnik weiter vorangeschritten war. Das einzig bekannte Original der Leibnizschen Rechenmaschine (um

1700) befindet sich in der Niedersächsischen Landesbibliothek in Hannover.

5. Automaten und Lochkartenmaschinen

Vor allem im 19. Jahrhundert gab es eine Reihe von technologischen Fortschritten, die sich indirekt auf die Weiterentwicklung der Rechenautomaten ausgewirkt haben. Hier ist vor allem die Entwicklung von programmgesteuerten Automaten zu nennen. Eingeleitet wurde diese Entwicklung durch Joseph- Marie Jacquard, der 1805 den automatischen Webstuhl erfand.

Es waren allerdings nicht nur die industriellen Einsatzmöglichkeiten, die die Entwicklung der Automaten vorantrieb. Es war auch die Begeisterung der damaligen Zeit für mechanisches Spielzeug und bei der begüterten Gesellschaft des 19. Jahrhunderts wurde es Mode, im Salon einen mechanischen Automaten aufzustellen. Meistens waren dies Puppen, die im Inneren eine kunstvolle Mechanik aufwiesen, durch die diese Puppen, angetrieben durch eine aufziehbare Feder, Bewegungen ausführen konnten. Bei einigen dieser Exemplare konnten unterschiedliche Bewegungen durch Lochkarten oder Lochscheiben gesteuert werden. Besonders schöne Exemplare findet man im Puppen- und Automatenmuseum in Monte Carlo. Diese Entwicklung wurde fortgesetzt durch die Musikautomaten, die ab Mitte des letzten Jahrhunderts in Musikhallen und Salons ihren Einsatz fanden.

Ihren ersten Einsatz für numerische Berechnungen fanden die Lochkarten in den, nach ihrem Erfinder Herman Hollerith benannten, Hollerith-Maschinen. Es waren elektrisch betriebene Zählmaschinen, bei denen die Dateneingabe über Lochkarten erfolgte. Damit waren diese Maschinen in der Lage, in sehr kurzer Zeit viele Daten statistisch auszuwerten. Ihre erste große Bewährungsprobe bestand diese Maschine bei der Volkszählung der USA im Jahre 1880. Sie wurden in den nächsten Jahren stetig verbessert und bald auch für vielfältige kaufmännische Rechenzwecke verwendet.

6. Die Rechenmaschinen von Babbage

Das Verdienst, als erster die Grundgedanken heutiger Rechenanlagen entworfen zu haben, gebührt Charles Babbage (1791 - 1871). Obwohl von ihm seine Maschinen nie komplett fertiggestellt wurden, lieferte er die entscheidenden Beiträge zum Übergang von einfachen Rechenmaschinen zu programmgesteuerten Rechenautomaten.

Die Aufgabe, mathematische Tabellen maschinell zu produzieren und mathematische Regeln, in Maschinen einzubetten, die sich Babbage 1821

stellte, beschäftige ihn sein gesamtes restliches Leben. Die oben aufgeführten Fehlerquellen waren ihm wohlbekannt und er schenkte viel Aufmerksamkeit der Eliminierung dieser Fehlerquellen. Seine Überlegungen zur Lösung waren die folgenden:

Da die Berechnungen von einer Maschine durchgeführt werden sollten, konnte dies theoretisch frei von Fehlern erfolgen, sofern die Maschine korrekt arbeitete. Da die Maschine auch über ein Druckwerk verfügen sollte, würden die Fehler des Kopiervorganges ebenfalls entfallen. Um den Druckvorgang fehlerfrei zu gestalten hat sich Babbage ein Sicherheitssystem überlegt. Er hat jeden Buchstaben mit einem bestimmten, individuellem Muster auf der Rückseite ausgestattet. Wenn nun alle Buchstaben eingespannt wurden, mußte ein Kontrolldraht durch die Buchstaben geschoben werden. Wenn dieser Draht blockierte, dann war ein Buchstabe falsch eingespannt, und man mußte diesen Fehler beheben, ansonsten konnte man nicht weiterarbeiten.

So war es möglich, auf einem Schlag, alle Fehlerquellen , die bis dahin zu Fehlern führten, zu beheben.

Babbage glaubte, daß seine Difference- Engine dieses leisten könne. Im Gegensatz zu den Maschinen von Schickard, Leibniz und Pascal war die Difference- Engine in der Lage, mehr als nur die vier Grundrechenarten durchzuführen. Vielmehr sollte diese Maschine automatisch Folgen von Funktionswerten ausgeben und diese anschließend ausdrucken können. Die Difference- Engine wurde so benannt, weil sie auf der Methode der finiten Differenzen basierte. Diese Methode war zu diesem Zeitpunkt wohl bekannt und wurde von den Kopfrechnern bei der Tabellenerstellung benutzt.

1823 beginnt Babbage mit dem staatlich geförderten Bau der Difference-Engine. Den Auftrag der Regierung erhielt er, nachdem er bis 1822 ein kleines Versuchsmodell einer Difference Engine fertiggestellt hatte, die lediglich aus sechs bis acht Ziffern bestand. Er beginnt mit der Entwicklung der Difference-Engine No.1, Babbages größtes Wagnis. Diese große Maschine benötigte 25.000 Teile und würde 8 Fuß hoch, 7 Fuß lang und 3 Fuß tief werden (2.4 x 2.1 x 0.9 m). Sie würde, sofern fertiggestellt, mehrere Tonnen wiegen.

Babbage heuerte Joseph Clement an, einem Werkzeugmacher und Zeichner. Die Kombination war zu damaligen Zeiten sehr geschätzt und kaum verbreitet. Dieser sollte Babbage die Maschine bauen. Die kommenden Jahre des Konstruierens, Entwicklens und Herstellens, waren die entäuschensten Jahre in Babbages Leben.

Die Arbeiten stoppten 1833 nach einem Streit mit Joseph Clement. Dieser machte von seinem Recht Gebrauch und nahm sämtliche Werkzeuge und die fähigsten Arbeiter mit. Mit der letzten Gehaltszahlung an Joseph Clement 1834, hatte die Regierung 17.470 Pfund, in den Bau der Difference- Engine

No1, investiert. Babbage selbst soll an die 20.000 Pfund investiert haben. Er bekam für seine Arbeit von der Regierung kein Gehalt, war aber durch das Erbe seines Vaters wohlhabend.

Um einen Vergleich hinsichtlich der bis dahin angefallenen Entwicklungskosten zu haben, seien die Kosten für den Bau der Lokomotive *John Bull*, von Robert Stephenson und Co. hergestellt und nach Amerika exportiert, angeführt. Sie betrugen an die 785 Pfund.

Die Meinung, wie nahe Babbage vor der Fertigstellung war, variieren. Fakt ist allerdings, daß essentielle Teile, für den Berechnungsmechanismus fertiggestellt wurden, und die und so die endgülige Realisierung realistisch war. Auf Babbages Instruktion hin, hat Clement 1832 ein kleinen Teil der Maschine fertiggestellt. Dieser Teil sollte für Demonstrationszwecke benutzt werden und umfaßte etwa ein siebtel der gesamten Maschine.

Ende des Jahre 1834 hatte Babbage eine noch ehrgeizigere Idee. Er träumte von der Analytical-Engine, einer revolutionären Maschine, die Babbage den Ruf eines Computerpioniers einbrachte. Wegen der Erfahrungen, die er beim Bau der Difference-Engine gemacht hatte, wollte Babbage, sofern er die Analytical-Engine bauen würde, dieses auf eigene Kosten machen. Er suchte nach Alternativen, um hunderte von annähernd gleiche Teile zu erstellen und suchte nach Methoden, die Kosten zu reduzieren. Nur ein Teil dieser Maschine wurde, zu seiner Lebenszeit hergestellt. Dieses Teil und ein weiteres Teil, was Babbages Sohn nach Tod seines Vaters hergestellt hatte, sowie einige experimentelle Montagesysteme, sind die einzigen physischen Realisierungen dieser Errungenschaft des 19 Jahrhunderts.

Bei der Entwicklung seiner Analytical-Engine, die mit Lochkarten, die aus der Webstuhltechnik kamen, wie ein Computer programmiert werden sollte, hatte Babbage so viele Erneuerungen und Verbesserungen gemacht, daß er von 1847 bis 1849 die Difference-Engine No.2 entwickelte. Diese Difference-Engine leistete das gleiche, wie ihr Vorgängermodell, allerdings wurde vieles vereinfacht. So hatte diese Maschine nur noch 4.000 Teile (mit Ausnahme des Druckmechanismus) und hatte eine Höhe von 7 Fuß, eine Länge von 11 Fuß und eine Tiefe von 18 Inch (2.1 x 3.4 x 0.5 m) und wog 3000 Kilogramm. Die Ausmaße der Analytical-Engine waren vergleichbar, mit einer kleineren Lokomotive (4.6 x 6.1 x 1.8 m). Da die Ausmaße dieser Maschine so gigantisch waren, hatte man vermutlich geplant, sie, mit Hilfe einer Dampfmaschine, anzutreiben.

Babbage bot die Konstruktionszeichnungen der Difference-Engine No.2 der Regierung an. Diese lehnte aber 1852 ab. Damit wurde auch diese Maschine nicht mehr zu Babbages Zeiten gebaut. Erst fast 150 Jahre später im Jahre 1991 wurde diese Maschine von zwei Ingenieuren, Reg Crick und Barrie Holloway, nachgebaut. Der Nachbau ist im Science Museum in London zu

besichtigen.

Von Babbage selbst sind nur wenige Beschreibungen über seine Maschine bekannt, aber es gibt eine Reihe von Beschreibungen von anderen Autoren. So nahm Babbage 1840 eine Einladung nach Turin an, wo er seine Pläne und Konzepte vorstellte. Seine Ausführungen wurden von L.F. Menabrea, einem jungen Ingenieur- Offizier, aufgezeichnet und 1843 veröffentlicht. Dieser Beitrag wurde von Augusta ADA, Countess of Lovelace, der Tochter von Lord Byron, ins Englische übersetzt. Lady Lovelace war so von den Ideen von Babbage begeistert, daß sie sogar eine Reihe von Programmen für die *Analytical Machine*, so z.B. für die Berechnung der Bernoulli-Zahlen entwarf. Sie wird daher oft als die erste Programmiererin angesehen werden und ihr zu Ehren wurde auch eine Programmiersprache, die im Auftrag des amerikanischen Verteidigungsministeriums entwickelt und heute noch vor allem im militärischen Bereich eingesetzt wird, benannt. Andererseits waren ihre Beiträge zur Programmierung wohl nicht so bedeutend, wie allgemein angenommen, denn die meisten Programme wurden durch Söhne von Babbage entwickelt. Dennoch hat eine erhebliche Mythisierung ihrer Person stattgefunden. Ihr Beitrag zur Entwicklung der Analytical Engine und ihr Verhältnis zu Babbage war stets Gegenstand umfangreicher Spekulationen. Sie reichten von „ Ada – die größte aller Huren in London“ bis zu einer von Babbages Ideen besessenen, die, als sie nicht mehr für ihn arbeiten konnte, „ ihre Existenz als öde und sinnlos“ empfand und daraufhin sowohl ihrer Liebes- wie auch ihrer Wettleidenschaft erlag. Umgekehrt soll Babbage die Lücke, die Adas Tod 1952 hinterlies, nur schwer oder gar nicht überwunden haben.

Babbage hat die *Analytical Machine* genau wie die *Difference Engine* selbst nie konstruktiv beendet. Zum einen lag es daran, daß die technischen Möglichkeiten der damaligen Zeit noch sehr beschränkt waren und zum anderen war er ein Perfektionist. Letzeres mag auch eine der Ursachen sein, warum die Arbeiten an der *Difference Engine* in Streitereien endeten. Er selbst schreibt 1835 über seine Arbeiten an der *Analytical Machine* an M. Quetelet, Mitglied der Königlichen Akademie der Wissenschaften in Brüssel:

„ The greatest difficulties of the invention are already overcome, but I shall need several more months to complete all the details and make the drawings.“

Er sollte sich irren, denn selbst 25 Jahre später war er immer noch nicht fertig. Nach seinem Tod 1871 setzte sein Sohn, Generalmajor Henry Babbage, seine Arbeiten fort. Er baute die zentrale arithmetische Einheit („ mill“) sowie die Ausgabeeinheit Eine Vorversion konnte 1878 vorgestellt werden; die endgültige Version war erstmalig am 21. Januar 1888 betriebsbereit und berechnete eine Tafel der Ergebnisse der Multiplikation von mit 1 bis 44.

Henry Babbage führte die von ihm gebauten Teile der *Analytical Machine* bis zum Beginn dieses Jahrhunderts auf verschiedenen Tagungen und Ausstellungen vor. Einige andere nahmen sich den Entwürfen seines Vaters an und konstruierten ähnliche Maschinen. Zu nennen sind vor allem Pery Ludgate, Torres y Quevedo und Louis Couffignal. Selbst Aiken hat sich mit der *Analytical Machine* befaßt, bevor er mit Unterstützung von IBM die *Harvard Mark 1* entwickelte, wie eine Veröffentlichung von ihm zeigt. Sein Studium der Arbeiten von Babbage war offensichtlich nicht sehr intensiv, da er sonst sicherlich z.B. das bereits von Babbage vorgesehene Konzept der bedingten Verzweigung realisiert hätte.

7. Die ersten programmierbaren Rechner

Den Verdienst, den ersten wirklich frei programmierbaren funktionsfähigen Rechner konstruiert zu haben, gebührt Konrad Zuse. Nach dem Studium und einer kurzen Tätigkeit als Statiker bei den Henschel-Flugzeug-Werken wandte er sich ab 1935, im Alter von 25 Jahren, dem Bau einer Rechenmaschine zu. Im Jahr 1936 waren die Konstruktionspläne für einen Rechner mit Gleitpunktarithmetik, der über ein gelochtes Filmband gesteuert werden konnte, fertig. Die Befehle waren 3-Adreßbefehle mit zwei Operanden- und einer Ergebnisadresse. Leider wurde seine erste Maschine, die *'Z1'*, nie vollständig funktionsfähig, da er erfahren mußte, daß die Mechanik für die von ihm verfolgten Ziele nicht flexibel genug war. Auch hatte er immer wieder Finanzierungsprobleme. Ein wesentlicher Durchbruch für die weiteren Arbeiten ergab sich aus der Zusammenarbeit mit Helmut Schreyer, einem Pionier des „elektronischen" Rechnens. Er erfand als Doktorand an der TH Berlin-Charlottenburg ab 1937 die Grundkomponenten zur Realisierung der Grundoperationen Konjunktion, Disjunktion und Negation sowie für Speicherelemente (Flip-Flops) auf der Basis von Röhren.

Schreyer erfand eine geschickte Kombination von Röhren und Glimmlampen, wobei die Röhren die Funktion der Wicklung eines elektromechanischen Relais und die Glimmlampen die Funktion der Kontakte übernahmen, und baute eine kleine Relaiskette auf. Diese Schaltung wurde 1938 im kleinen Kreis der Technischen Hochschule vorgeführt und die Vision einer elektronischen Rechenanlage erläutert. Da die größten elektronischen Geräte der damaligen Zeit Sendeanlagen mit einigen hundert Röhren waren, erzeugte die Idee, eine Rechenmaschine mit zweitausend Röhren und einigen tausend Glimmlampen zu bauen, nur Kopfschütteln.

Hierdurch ernüchtert, plante Zuse den Bau einer Relaismaschine. Eine finanzielle Unterstützung bekam er nun durch Dr. Kurt Pennke, einem Fabrikanten von Tischrechenmaschinen. Das zweite Gerät, die *'Z2'*, setzte sich aus dem mechanischen 16-Wort-Speicher der *'Z1'* und einem neuen, mit elektromagnetischen Relais aufgebauten Rechenwerk zusammen. Das Gerät

war 1940 vorführbereit und wurde der Deutschen Versuchsanstalt für Luftfahrt in Berlin-Adlershof erfolgreich vorgeführt. Bemerkenswerterweise war dies praktisch der einzige erfolgreiche Einsatz der *'Z2'*. Das dauernde Versagen hatte einen einfachen Grund: Zuse hatte in seiner Materialnot alte Telefonrelais benutzt und war daher gezwungen gewesen, Ruhekontakte zu Arbeitskontakten umzubauen. Er hatte jedoch übersehen, daß die oberen Kontaktfedern eine Auflage brauchten, um die nötige Vorspannung für den Kontaktdruck zu erwirken.

Diese Vorführung der *'Z2'* hatte genügt, die Deutsche Versuchsanstalt für Luftfahrt zu veranlassen, die *'Z3'* mitzufinanzieren. Sie war 1941 fertig und das erste Gerät, das wirklich voll funktionsfähig alle wichtigen Elemente einer programmgesteuerten Rechenmaschine enthielt. Die *Z3* wurde während des Krieges mehreren Dienststellen vorgeführt; sie wurde indes nie im Routinebetrieb eingesetzt. Sie wurde 1944 im Bombenkrieg zerstört und 1960 nachgebaut und im Deutschen Museum in München aufgestellt.

1942 begann Zuse mit dem Bau der *Z4*, einer Weiterentwicklung der *Z3*. Auch die *Z4* war noch voll auf die Elektromechanik abgestellt, wie es dem damaligen Stand der Technik entsprach. Für das Speicherwerk empfahl sich die mechanische Konstruktion; Rechenwerk und Steuerungen wurden mit Relais und Schrittschaltern aufgebaut. Um dem Gerät von der Programmierseite her eine größere Flexibilität zu geben, wurden mehrere Ausbaustufen mit mehreren Abtastern und Lochern vorgesehen. Die Arbeiten an der *Z4* wurden schon stark durch den Bombenkrieg behindert. Die *Z4* mußte während des Krieges innerhalb Berlins dreimal ihren Platz wechseln.

Ende 1944 stand die *Z4* kurz vor ihrer Vollendung, als kriegsbedingt ein Weiterarbeiten in Berlin nicht mehr möglich war. Die *Z4* wurde mit dem Zug nach Göttingen transportiert, wobei sie mit viel Glück mehrere Bombenangriffe überstand. Der Abtransport aus Berlin war nur möglich, weil die damalige Bezeichnung der Maschine nicht *Z4*, sondern *V4*[1] lautete. Durch den Gleichklang dieser Abkürzung mit der für die sog. Vergeltungswaffen *V1* und *V2* und der von seinem Mitarbeiter, Dr. Funk erfundenen Parole „ Die *V4* muß aus Berlin in Sicherheit gebracht werden“ , konnten die Behörden über den wahren Inhalt der Fracht getäuscht werden. In Göttingen, in den Räumen der Aerodynamischen Versuchsanstalt, konnte die *Z4* dann fertiggestellt werden. Danach wurde sie vor den anrückenden Engländern nach Hinterstein im Allgäu in Sicherheit gebracht und in dem Keller eines Hinterhauses versteckt. Obwohl sowohl die Franzosen als auch die Engländer nach ihr suchten, blieb sie unentdeckt. Bis zur Währungsreform 1948 ruhten die Arbeiten an der *Z4*. Zwischenzeitlich war Zuse 1946 von Hinterstein nach Hopferau bei Füssen umgezogen, wo er die *Z4* in einem ehemaligen

[1]

Pferdestall unterbrachte.

Eines Tages - es war im Jahr 1949 - tauchte ein vornehmer Wagen aus der Schweiz in Hinterstein auf. Professor Stiefel von der Eidgenössischen Technischen Hochschule Zürich war zu Ohren gekommen, daß irgendwo in einem kleinen Dorf im Allgäu ein Computer zu finden sei. Er war eben von einer Studienreise in die USA zurückgekommen, wo er „ viele schöne Maschinen in schöne Schränken mit Chromleisten“ gesehen hatte. Der Professor war nicht wenig überrascht, als er die äußerlich doch schon ein wenig ramponierte *Z4* auch noch in einem Pferdestall aufgebaut fand. Trotzdem diktierte er Zuse eine einfache Differentialgleichung, die Zuse sofort programmieren, auf der Maschine vorführen und lösen konnte. Danach schloss er mit Zuse einen Vertrag: die *Z4* sollte - nach gründlicher Überholung und Reinigung - an die ETH ausgeliehen werden.

1950 wurde die *Z4* verladen und nach Zürich geschafft. Es war ihr sechster Transport. Zur feierlichen Inbetriebnahme der *Z4* noch im selben Jahr waren etwa hundert Gäste aus Industrie und Wissenschaft geladen. Alles war gut vorbereitet; die Maschine hatte vormittags ihre Testläufe gemacht, nachmittags um vier sollte die Vorführung stattfinden. Nach dem Mittagessen aber bockte die Maschine plötzlich und sprühte an den unglaublichsten Stellen Funken. Kurzschlüsse brannten ganze Leitungen durch. Nichts, aber auch nichts funktionierte mehr. Es begann ein großes Rätselraten. Prof. Stiefel, der mit seinen Mitarbeitern Rutishauser und Speiser für das *Z4*-Projekt verantwortlich war, blieb äußerlich ruhig; aber im Geiste sah er sich gewiß schon gründlich blamiert. Man darf nicht vergessen, daß damals einiger Mut dazu gehörte, einen Computer ausgerechnet aus Deutschland kommen zu lassen. Zuse suchte eine gute Stunde, dann hatte er den Fehler gefunden: Das Gerät hatte für Ansprech- und Haltekreise zwei verschiedene Spannungsniveaus, sechzig und achtundvierzig Volt, und man hatte einen neuen Umformer in Betrieb genommen, der diese Spannungen liefern sollte. Leider hatte man dabei nicht beachtet, daß die Polung beim Einschalten des Umformers willkürlich erfolgte, und zwar unabhängig für beide Spannungen. So konnten an Stellen, an denen sonst nur zwölf Volt Spannungsdifferenz herrschten, plötzlich einhundertacht Volt Spannung auftreten. Das hatte nicht gutgehen können. Ihm blieb genau eine halbe Stunde Zeit, den Fehler abzustellen und die durchgebrannten Leitungen zu ersetzen. Er schaffte es; der leicht brenzlige Geruch wurde durch Lüften beseitigt und um sechzehn Uhr waren die illustren Gäste Zeugen einer einwandfreien Vorführung und die *Z4* nahm in Zürich ihren Betrieb auf. Die *Z4* arbeitete mit der Zeit so zuverlässig, daß man sie nachts unbewacht durchlaufen ließ.

Nach fünfjähriger Arbeit in Zürich übersiedelte die *Z4* noch einmal in ein französisch- deutsches Forschungsinstitut in Saint Louis und blieb dort

weitere fünf Jahre in Betrieb. Für die ETH Zürich entwickelten Stiefel, Speiser und Rutishauser einen eigenen Computer, die ERMETH.

Damit ist dieser kleine Überblick über die Geschichte der Entwicklung von Rechenautomaten abgeschlossen. Wer sich intensiver mit dieser Materie beschäftigen möchte, kann weitere Details und insbesondere zahlreiche Abbildungen auf der Webseite meines Lehrstuhls

wwwmath1.uni-muenster.de:8000/info/Professoren/Lippe/lehre/skripte/index.html

nachlesen.

Viel Spass

From Communication Histories to State Transition Machines

Walter Dosch

Institute of Software Technology and Programming Languages
University of Lübeck
Lübeck, Germany

Abstract. The black-box view of an interactive component concentrates on the input/output behaviour based on communication histories. The glass-box view discloses the component's internal state with inputs effecting an update of the state. The black-box view is modelled by a stream processing function, the glass-box view by a state transition machine. We present a formal method for transforming a stream processing function into a state transition machine with input and output. We introduce states as abstractions of the input history and derive the machine's transition functions using history abstractions. The state refinement is illustrated with two applications, viz. iterator components and an interactive stack.

Keywords Interactive component, stream processing, state transition machine, communication history, history abstraction, iterator component, interactive stack

1 Introduction

A distributed system consists of a network of components that communicate asynchronously via unidirectional channels. The communication histories are modelled by sequences of messages, called streams. Streams abstract from discrete or continuous time, since they record only the succession of messages. The input/output behaviour of a communicating component is described by a stream processing function [9, 10] mapping input histories to output histories.

During the development of a component, the software designer employs different points of view. On the specification level, a component is considered as a black box whose behaviour is determined by the relation between input and output histories. The external view is relevant for the service provided to the environment.

On the implementation level, the designer concentrates on the component's internal state where an input is processed by updating the internal state. The internal view, also called glass-box view, is described by a state transition machine with input and output.

A crucial design step amounts to transforming the specified behaviour of a communicating component into a state-based implementation. In our approach,

W. Goerigk (Hrsg.): Programmiersprachen und Rechenkonzepte. Bericht Nr. 0410, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

we conceive machine states as abstractions of the input history. The state stores information about the input history that influences the component's output on future input. In general, there are different abstractions of the input history which lead to state spaces of different granularity.

This paper presents a formal method, called *state refinement*, for transforming stream processing functions into state transition machines. The transformation is grounded on history abstractions which identify subsets of input histories as the states of the machine. The state refinement preserves the component's input/output behaviour, if we impose two requirements. Upon receiving further input, a history abstraction must be compatible with the state transitions and with the generation of the output stream. The formal method supports a top-down design deriving the state-based implementation from a behavioural specification in a safe way.

The paper is organized as follows. In Section 2 we summarize the basic notions for the functional description of interactive components with communication histories. Section 3 introduces state transition machines with input and output. Section 4 presents the systematic construction of a state transition machine that implements a stream processing function in a correct way. History abstractions relate input histories to machine states. With their help, the transition functions of the machine can be derived involving the output extension of the stream processing function. In the subsequent sections, we demonstrate the state refinement for different types of applications. In Section 5 , the transformation of iterator components leads to state transition machines with a trivial state space resulting from the constant history abstraction. Section 6 discusses the state-based implementation of an interactive stack. The history abstraction leading to a standard implementation results from combining a control state and a data state in a suitable way.

2 Streams and Stream Processing Functions

In this section we briefly summarize the basic notions about streams and stream processing functions to render the paper self-contained. The reader is referred to [18] for a survey and to [19] for a comprehensive treatment.

2.1 Finite Streams

Streams model the communication history of a channel which is determined by the sequence of data transferred via a channel. Untimed streams record only the succession of messages and provide no further information about the timing.

Given a non-empty set $\mathcal{A}$ of data, the set $\mathcal{A}^\star$ of finite *communication histories*, for short *streams*, over $\mathcal{A}$ is the least set with respect to subset inclusion defined by the recursion equation $\mathcal{A}^\star = \{\langle\rangle\} \cup \mathcal{A} \times \mathcal{A}^\star$. A stream is either the *empty stream* $\langle\rangle$ or is constructed by the operation $\triangleleft : \mathcal{A} \times \mathcal{A}^\star \to \mathcal{A}^\star$ attaching an element to the front of a stream. We denote streams by capital letters and

elements of streams by small letters. A stream $X = x_1 \lhd x_2 \lhd \ldots \lhd x_n \lhd \langle\rangle$ $(n \geq 0)$ is denoted by $\langle x_1, x_2, \ldots, x_n\rangle$ for short.

The *concatenation* $X\&Y$ of two streams $X = \langle x_1, x_2, \ldots, x_k\rangle$ and $Y = \langle y_1, y_2, \ldots, y_l\rangle$ over the same set $\mathcal{A}$ of data yields the stream $\langle x_1, x_2, \ldots, x_k, y_1, y_2, \ldots, y_l\rangle$ of length $k+l$. The concatenation $X\&\langle x\rangle$ appending an element x at the rear of a stream X is abbreviated as $X \rhd x$.

2.2 Prefix Order

Operational progress in time is modelled by the prefix order. The longer stream forms an extension of the shorter history, and, vice versa, the shorter stream is an initial history of the longer stream.

A stream X is called a *prefix* of a stream Y, denoted $X \sqsubseteq Y$, iff there exists a stream R with $X\&R = Y$. The set of finite streams together with the prefix relation forms a partial order with the empty stream as the least element. Monotonic functions on finite streams possess unique continuous extensions to infinite streams [13].

2.3 Stream Processing Functions

The history of data passing along a communication channel between components and, possibly, their environment is mathematically captured by the notion of a stream. Thus, a deterministic component which continuously processes data from its input ports and emits data at its output ports can be considered as a function mapping input histories to output histories.

A *stream processing function* $f : \mathcal{A}^\star \to \mathcal{B}^\star$ maps an input stream to an output stream. The input type $\mathcal{A}$ and the output type $\mathcal{B}$ determine the syntactic *interface* of the component.

We require that a stream processing function is *monotonic* with respect to the prefix order: $f(X) \sqsubseteq f(X\&Y)$. This property ensures that a prolongation of the input history leads to an extension of the output history. A communicating component cannot change the past output when receiving future input.

A stream processing function describes the *(input/output) behaviour* of a component.

2.4 Output Extension

A stream processing function summarizes the behaviour of a component on entire input streams. A finer view reveals the causal relationship between single elements in the input stream and corresponding segments of the output stream.

The output extension isolates the effect of an input on the output stream after processing a prehistory.

Definition 1 *The* output extension $\varepsilon_f : \mathcal{A}^\star \times \mathcal{A} \to \mathcal{B}^\star$ *of a stream processing function* $f : \mathcal{A}^\star \to \mathcal{B}^\star$ *is defined by*

$$f(X \rhd x) = f(X) \;\&\; \varepsilon_f(X, x) \;\; . \tag{1}$$

The output extension completely determines the behaviour of a stream processing function apart from its result for the empty input.

3 State Transition Machines with Input and Output

The operational behaviour of distributed systems is often formalized by labelled state transition systems specifying a transition relation between states associated with labels [21]. The transitions denote memory updates, inputs, outputs, or other actions. For the purposes of modelling communicating components, we associate a state transition with receiving an element on the input channel and sending data to the output channel.

3.1 Architecture of the Machine

A state transition machine reacts on input with an update of the internal state generating a sequence of outputs.

Definition 2 *A* state transition machine with input and output*, for short a* state transition machine*,* $M = (\mathcal{Q}, \mathcal{A}, \mathcal{B}, next, out, q_0)$ *consists of a non-empty set* $\mathcal{Q}$ *of* states*, a non-empty set* $\mathcal{A}$ *of* input data*, a non-empty set* $\mathcal{B}$ *of* output data*, a* (single-step) state transition function $next : \mathcal{Q} \times \mathcal{A} \to \mathcal{Q}$*, a* (single-step) output function $out : \mathcal{Q} \times \mathcal{A} \to \mathcal{B}^\star$*, and an* initial state $q_0 \in \mathcal{Q}$*. The types* $\mathcal{A}$ *and* $\mathcal{B}$ *determine the* interface *of the state transition machine.*

Given a current state and an input, the single-step state transition function determines a unique successor state. The single-step output function yields a finite sequence of elements, not just a single element.

The single-step functions can naturally be extended to finite input streams.

Definition 3 *The* multi-step state transition function $next^\star : \mathcal{Q} \to [\mathcal{A}^\star \to \mathcal{Q}]$ *yields the state reached after processing a finite input stream:*

$$next^\star(q)(\langle\rangle) = q \tag{2}$$

$$next^\star(q)(x \lhd X) = next^\star(next(q, x))(X) \tag{3}$$

The multi-step output function $out^\star : \mathcal{Q} \to [\mathcal{A}^\star \to \mathcal{B}^\star]$ *accumulates the output stream for a finite input stream:*

$$out^\star(q)(\langle\rangle) = \langle\rangle \tag{4}$$

$$out^\star(q)(x \lhd X) = out(q, x) \,\&\, out^\star(next(q, x))(X) \tag{5}$$

The multi-step output function describes the (input/output) behaviour *of the state transition machine.*

For each state $q \in \mathcal{Q}$, the multi-step output function $out^\star(q) : \mathcal{A}^\star \to \mathcal{B}^\star$ constitutes a stream processing function. It abstracts from the state transitions and offers a history-based view of the component.

3.2 Output Equivalence

We aim at transforming a state transition machine into a more compact one with a reduced number of states without changing the behaviour. To this end, we are interested in states which induce an equal behaviour when the state transition machine receives further input.

Definition 4 *Two states $p, q \in \mathcal{Q}$ of a state transition machine $M = (\mathcal{Q}, \mathcal{A}, \mathcal{B}, next, out, q_0)$ are called* output equivalent*, denoted $p \approx q$, iff they generate the same output for all input streams: $out^\star(p) = out^\star(q)$.*

An observer of the state transition machine cannot distinguish output equivalent states, as they produce the same output stream for every input stream.

Proposition 1 *Successor states of output equivalent states are also output equivalent.*

3.3 Related models

State transition machines with input and output are closely related to a variety of state-based computing devices used to specify, verify, and analyse the behaviour of distributed systems, among others *generalized sequential machines* [7], *port input/output automaton* [11, 12], *Stream X-machines* [5] and X-machines [7], Harel's *statecharts* [8], μ-Charts [16] and *UML state diagrams* [15, 14]. A different type of *state transition systems* were used in [1] for specifying the behaviour of components and, in particular, for the verification of safety and liveness properties [3].

4 From Stream Processing Functions to State Transition Machines

In this section, we implement stream processing functions by state transition machines using history abstractions. Given a stream processing function, we construct a state transition machine with the same interface and the same behaviour. The crucial design decision amounts to choosing an appropriate set of states. In our approach, the states of the machine represent sets of input histories that have the same effect on the output for all future input streams.

4.1 History Abstractions

A history abstraction extracts from an input history certain information that influences the component's future behaviour.

Definition 5 *For a stream processing function $f : \mathcal{A}^\star \to \mathcal{B}^\star$ and a set $\mathcal{Q}$ of* states*, a function $\alpha : \mathcal{A}^\star \to \mathcal{Q}$ is called a* history abstraction *for f, if it is* output compatible *(6) and* transition closed *(7):*

$$\alpha(X) = \alpha(Y) \Longrightarrow \varepsilon_f(X, x) = \varepsilon_f(Y, x) \tag{6}$$

$$\alpha(X) = \alpha(Y) \Longrightarrow \alpha(X \triangleright x) = \alpha(Y \triangleright x) \tag{7}$$

The output compatibility guarantees that a history abstraction identifies at most those input histories which have the same effect on future output. The transition closedness ensures that extensions of identified streams are identified as well:

$$\alpha(X) = \alpha(Y) \Longrightarrow \alpha(X\&Z) = \alpha(Y\&Z) \tag{8}$$

The transition closedness constitutes a general requirement, whereas the output compatibility refers to the particular stream processing function.

4.2 Construction of the State Transition Machine

When implementing a stream processing function with a state transition machine, the history abstraction determines the state space, the transition functions, and the initial state.

Definition 6 *Given a stream processing function $f : \mathcal{A}^\star \to \mathcal{B}^\star$ and a surjective history abstraction $\alpha : \mathcal{A}^\star \to \mathcal{Q}$ for f, we construct a state transition machine $M[f, \alpha] = (\mathcal{Q}, \mathcal{A}, \mathcal{B}, next, out, q_0)$ with the same interface as follows:*

$$next(\alpha(X), x) = \alpha(X \triangleright x) \tag{9}$$

$$out(\alpha(X), x) = \varepsilon_f(X, x) \tag{10}$$

$$q_0 = \alpha(\langle\rangle) \tag{11}$$

The state transition function and the output function are well-defined, since the history abstraction is surjective, transition closed, and output compatible.

The following proposition establishes the correctness of the implementation step.

Proposition 2 *Under the assumptions of Def. 6 , the stream processing function and the multi-step output function of the state transition machine agree:*

$$f(X) = f(\langle\rangle)\ \&\ out^\star(q_0)(X) \tag{12}$$

In particular, for a strict stream processing function we have $f = out^\star(q_0)$.

In general, a stream processing function possesses various history abstractions identifying different subsets of input histories as states.

The finest history abstraction is given by the identity function $\alpha(X) = X$ identifying no input histories at all. The associated state transition machine is called the *canonical state transition machine*. Its states correspond to input histories, the state transition function extends the input history input by input, the output function is the output extension.

The coarsest history abstraction $\alpha(X) = [X]_\approx$ maps every input history to the class of output equivalent input histories. The associated state transition machine possesses a minimal state space.

We can generalize the construction of the state transition machine to history abstraction functions that are not surjective. In this case, the state transition functions are uniquely specified only on the subset of reachable states. The transition functions can be defined in an arbitrary way on the subset of unreachable states; this will not influence the input/output behaviour of the machine starting in the initial state.

4.3 State Refinement

Every stream processing function can be transformed into a state transition machine with the same input/output behaviour using a history abstraction.

This universal construction lays the foundations for a formal method for developing a correct state-based implementation of a communicating component from its input/output-oriented specification. We call the formal method *state refinement*, since it transforms a component's communication-oriented black-box description into a state-based glass-box description. The history abstraction documents the essential design decisions for the state space. The state refinement complements other methods of refinement for communicating components, among others interface refinement [2] , property refinement [4], and architecture refinement [17] .

We presented the state refinement transformation $f \mapsto M[f, \alpha]$ for unary stream processing functions f only. The transformation generalizes to stream processing functions with several arguments in a natural way [19] .

5 History Independent Components

This section applies the state refinement transformation to the class of components whose behaviour does not dependent on the previous input history. We uniformly describe the set of history independent stream processing functions by a higher-order function. A constant history abstraction leads to an associated state transition machine with a singleton set as state space.

5.1 Iterator Components

An iterator component repeatedly applies a basic function to all elements of the input stream.

Iterator components are uniformly described by the higher-order function $map : [\mathcal{A} \to \mathcal{B}^\star] \to [\mathcal{A}^\star \to \mathcal{B}^\star]$ with

$$map(g)(\langle\rangle) = \langle\rangle \tag{13}$$

$$map(g)(x \triangleleft X) = g(x) \,\&\, map(g)(X) \ . \tag{14}$$

The higher-order function *map* concatenates the subsequences generated by the single input elements to form the output stream. For every basic function g , the function $map(g)$ distributes over concatenation. Therefore the function $map(g)$ is prefix monotonic. The output extension $\varepsilon_{map(g)}(X, x) = g(x)$ depends only on the current input, but not on the previous input history.

5.2 State Transition Machine of an Iterator Component

The history abstraction of an iterator component need not preserve any information of the previous input history. Thus any transition closed function forms a proper history abstraction, in particular, any constant function.

For constructing the state transition machine $M[map(g), const]$, we choose a singleton state space $\mathcal{Q} = \{q_0\}$ and a constant history abstraction $const(X) = q_0$. The resulting state transition machine is shown in Fig. 1.

$M[map(g), const] = (\{q_0\}, \mathcal{A}, \mathcal{B}, next, out, q_0)$
$next(q_0, x) = q_0$
$out(q_0, x) = g(x)$

Fig. 1. State transition machine of an iterator component

The history independent behaviour of an iterator component is reflected by a "state-free" machine whose singleton state is irrelevant.

Vice versa, any state transition machine $M = (\{q_0\}, \mathcal{A}, \mathcal{B}, next, out, q_0)$ with a singleton state implements the behaviour of an iterator component $map(g)$ where the basic function $g : \mathcal{A} \to \mathcal{B}^\star$ is defined as $g(x) = out(q_0, x)$.

Iterator components are frequently used in various application areas, among others in transmission components, processing units, and control components.

6 Interactive Stack

As a final application we construct the implementation of an interactive stack. The application shows how to combine a control abstraction and a data abstraction into an overall history abstraction.

6.1 Specification

An interactive stack is a communicating component that stores and retrieves data following a last-in/first-out strategy. The component reacts on requests outputting the last datum which has previously been stored, but was not requested yet.

The interactive stack is fault-sensitive: after a pop command to the empty stack, the component breaks and provides no further output whatsoever future input arrives.

Let $\mathcal{D}$ denote the non-empty set of data to be stored in the stack. The component's input $\mathcal{I} = \{pop\} \cup push(\mathcal{D})$ consists of pop commands or push commands along with the datum to be stored.

The component's behaviour forms a stream processing function $stack : \mathcal{I}^\star \to \mathcal{D}^\star$ defined by the following equations ($P \in push(\mathcal{D})^\star$):

$$stack(P) = \langle\rangle \tag{15}$$

$$stack(P \& \langle push(d), pop\rangle \& X) = d \triangleleft stack(P \& X) \tag{16}$$

$$stack(pop \triangleleft X) = \langle\rangle \tag{17}$$

A sequence of push commands generates no output (15). A pop command outputs the datum stored most recently (16). After an erroneous pop command, the interactive stack breaks (17).

The behaviour of the interactive stack leads to the output extension $\varepsilon_{stack} : \mathcal{I}^\star \times \mathcal{I} \to \mathcal{D}^\star$ defined by $(P \in push(\mathcal{D})^\star)$:

$$\varepsilon_{stack}(X, push(d)) = \langle\rangle \tag{18}$$
$$\varepsilon_{stack}(P \triangleright push(d), pop) = \langle d\rangle \tag{19}$$
$$\varepsilon_{stack}(\langle\rangle, pop) = \langle\rangle \tag{20}$$
$$\varepsilon_{stack}(pop \triangleleft X, pop) = \langle\rangle \tag{21}$$
$$\varepsilon_{stack}(P\&\langle push(d), pop\rangle\&X, pop) = \varepsilon_{stack}(P\&X, pop) \tag{22}$$

A push command generates no output after any input history (18). A pop command yields the datum stored most recently which was not requested yet (19) unless the stack contains no datum (20,21).

6.2 Control Abstraction

The future behaviour of a fault-sensitive stack is influenced by the occurrence of an illegal pop command in the preceding input history.

We discriminate between regular and erroneous input histories using a binary control state $Control = \{reg, err\}$. The control abstraction $control : \mathcal{I}^\star \to Control$ classifies input histories as regular or erroneous $(P \in push(\mathcal{D})^\star)$:

$$control(P) = reg \tag{23}$$
$$control(P\&\langle push(d), pop\rangle\&X) = control(P\&X) \tag{24}$$
$$control(pop \triangleleft X) = err \tag{25}$$

A sequence of push commands forms a regular input history (23), whereas a pop command without a preceding push command gives rise to an erroneous input history (25).

The control abstraction is neither transition closed nor output compatible, since it identifies all regular input histories, but forgets the data stored in the component.

6.3 Data Abstraction

The future behaviour of the interactive stack will be influenced by the collection of data stored in the component from the previous input history.

As a second abstraction, we explore the state $Data = \mathcal{D}^\star$ storing a stack of data. The data abstraction $data : \mathcal{I}^\star \to Data$ extracts from the input history the stack of data retained in the component after processing the input stream $(n \geq 0)$:

$$data(\langle push(d_1), \ldots, push(d_n)\rangle) = \langle d_1, \ldots, d_n\rangle \tag{26}$$
$$data(P\&\langle push(d), pop\rangle\&X) = data(P\&X) \tag{27}$$
$$data(pop \triangleleft X) = \langle\rangle \tag{28}$$

The data abstraction is neither output compatible nor transition closed. It identifies regular input histories leading to the empty stack with erroneous input histories resulting in a broken stack.

6.4 History Abstraction

We integrate the control abstraction and the data abstraction into a joint history abstraction.

This design decision leads to a composite state space $\mathcal{Q} = \mathit{Control} \times \mathit{Data}$ combining a control part and a data part. The abstraction function $\alpha : \mathcal{I}^\star \to \mathit{Control} \times \mathit{Data}$ pairs the control and the data abstraction.

The abstraction function : $\alpha(X) = (\mathit{control}(X), \mathit{data}(X))$ keeps all required information from the input history which determines the component's future behaviour. The abstraction function is indeed a history abstraction and supports the transition to a state-based implementation.

6.5 State Transition Machine of an Interactive Stack

The implementation of the interactive stack is derived from the input/output behaviour using the combined history abstraction for control and data states.

The resulting state transition machine is summarized in Fig. 2 . In a regular

$M[\mathit{stack}, \alpha] = (\mathit{Control} \times \mathit{Data}, \mathcal{I}, \mathcal{D}, \mathit{next}, \mathit{out}, (\mathit{reg}, \langle\rangle))$
$\mathit{next}((\mathit{reg}, Q), \mathit{push}(d)) = (\mathit{reg}, Q \triangleright d)$
$\mathit{next}((\mathit{reg}, Q \triangleright q), \mathit{pop}) = (\mathit{reg}, Q)$
$\mathit{next}((\mathit{reg}, \langle\rangle), \mathit{pop}) = (\mathit{err}, \langle\rangle)$
$\mathit{next}((\mathit{err}, \langle\rangle), x) = (\mathit{err}, \langle\rangle)$
$\mathit{out}((\mathit{reg}, Q), \mathit{push}(d)) = \langle\rangle$
$\mathit{out}((\mathit{reg}, Q \triangleright q), \mathit{pop}) = \langle q \rangle$
$\mathit{out}((\mathit{reg}, \langle\rangle), \mathit{pop}) = \langle\rangle$
$\mathit{out}((\mathit{err}, \langle\rangle), x) = \langle\rangle$

Fig. 2. State transition machine of an interactive stack

state, a push command attaches an element to the stack and produces no output. Moreover, a pop command delivers the top of a non-empty stack; for an empty stack it leads to the error state. This state cannot be left any more by further input which produces no output in the error state.

The subset of states reachable from the initial state $(\mathit{reg}, \langle\rangle)$ is isomorphic to the direct sum of the data stack and an error state:

$$\{\mathit{reg}\} \times \mathcal{D}^\star \cup \{(\mathit{err}, \langle\rangle)\} \;\simeq\; \mathcal{D}^\star + \{\mathit{err}\} \tag{29}$$

The transition functions defined on the subset of reachable states can simply be extended to the subset of unreachable states by setting $\mathit{next}((\mathit{err}, Q), x) = \mathit{err}$ and $\mathit{out}((\mathit{err}, Q), x) = \langle\rangle$.

6.6 State Transition Table of an Interactive Stack

For practical purposes, state transition machines are often described by *state transition tables* displaying the different transition rules in a clear way.

Fig. 3 describes the interactive stack by a state transition table. The four *transition rules* relate current states and inputs to new states and outputs. The transition rules tabulate the transition functions *next* and *out* . We use the notational convention that the constituents of the successor state are designated by a prime. For an empty input stream, the state transition table produces no output which agrees with Equation (15) .

Control	Data	Input	Control'	Data'	Output
reg	Q	$push(d)$	*reg*	$Q \triangleright d$	$\langle\rangle$
reg	$Q \triangleright q$	*pop*	*reg*	Q	$\langle q \rangle$
reg	$\langle\rangle$	*pop*	*err*	$\langle\rangle$	$\langle\rangle$
err	$\langle\rangle$	x	*err*	$\langle\rangle$	$\langle\rangle$

Fig. 3. State transition table of an interactive stack

7 Conclusion

Nowadays the specification and the systematic design of communicating components belongs to the central challenges of modern software technology. The software design must safely bridge component descriptions on different levels of abstraction.

The component's specification reflects a communication-oriented view concentrating on input and output histories. History-based specifications raise the abstraction level of initial descriptions. The black-box view provides a functional model of the component important for constructing networks in a compositional way.

The component's implementation decisively depends on the internal state supporting an efficient realization of the transition functions. The glass-box view discloses the component's internal state which is in general composed from various control and data parts.

This paper contributes to a better understanding how to relate communication-oriented and state-based descriptions. We presented a formal method for systematically transforming a stream processing function into a state transition machine. The state refinement employs history abstractions to bridge the gap between input histories and machine states. The transition functions can be derived from the defining equations using the Lübeck Transformation System [6] .

Yet, the crucial design decision consists in discovering a suitable history abstraction which determines the state space. In general, the state of a component

must store at least the information which is needed to process further inputs in a correct way. The particular information depends on the area of application. For example, the state of a counter records the sum of all elements which passed the component; so it depends on the entire prehistory. The state of a memory cell remembers the datum of the last write command which is the only decisive event in the prehistory. The state of a shift register stores a final segment of the input stream that is withheld from the output stream. The state of a transmission component may record the active channel, the successful transmission or the failure of acknowledge.

The state refinement presents a standard transformation from a denotational to an operational description of interactive components [20] . The refinement step can be prepared by calculating the output extension of the stream processing function. This step localizes the component's reaction in response to a single input wrt. a previous input history.

Among the candidates for an implementation, we identified the canonical state transition machine whose state records the complete input history. State transition machines with a reduced state space originate from the canonical machine by identifying states as input histories under history abstractions. By construction, all resulting state transition machines correctly implement the specified behaviour.

The history-oriented and the state-based description of software or hardware components allow complementary insights. Both formalisms shows advantages and shortcomings with respect to compositionality, abstractness, verification, synthesis, and tool support. In long term, proven design methods must flexibly bridge the gap between functional behaviour and internal realization following sound refinement rules.

Origin of this Summary

This summary is an excerpt of the forthcoming paper *Transforming Stream Processing Functions into State Transition Machines* by W. Dosch and A. Stümpel to appear in the *Journal of Computational Methods in Sciences and Engineering*.

References

1. M. Breitling and J. Philipps. Diagrams for dataflow. In J. Grabowski and S. Heymer, editors, *Formale Beschreibungstechniken für verteilte Systeme*, pages 101–110. Shaker Verlag, 2000.
2. M. Broy. (Inter-)action refinement: The easy way. In M. Broy, editor, *Program Design Calculi*, volume 118 of *NATO ASI Series F*, pages 121–158. Springer, 1993.
3. M. Broy. From states to histories: Relating state and history views onto systems. In T. Hoare, M. Broy, and R. Steinbrüggen, editors, *Engineering Theories of Software Construction*, volume 180 of *Series III: Computer and System Sciences*, pages 149–186. IOS Press, 2001.

4. M. Broy and K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Monographs in Computer Science. Springer, 2001.
5. T. Bălănescu, A. J. Cowling, H. Georgescu, M. Gheorghe, M. Holcombe, and C. Vertan. Communicating stream X-machines systems are no more than X-machines. *Journal of Universal Computer Science*, 5(9):494–507, 1999.
6. W. Dosch and S. Magnussen. The Lübeck Transformation System: A transformation system for equational higher order algebraic specifications. In M. Cerioli and G. Reggio, editors, *Recent Trends in Algebraic Development Techniques (WADT 2001)*, number 2267 in Lecture Notes in Computer Science, pages 85–108. Springer, 2002.
7. S. Eilenberg. *Automata, Languages, and Machines*, volume A. Academic Press, 1974.
8. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
9. G. Kahn. The semantics of a simple language for parallel programming. In J. Rosenfeld, editor, *Information Processing 74*, pages 471–475. North–Holland, 1974.
10. G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing 77*, pages 993–998. North–Holland, 1977.
11. N. Lynch and M. R. Tuttle. An introduction to input/output automata. *Centrum voor Wiskunde en Informatica, Amsterdam, CWI-Quarterly*, 2(3):219–246, Sept. 1989.
12. N. A. Lynch and E. W. Stark. A proof of the Kahn principle for input/output automata. *Information and Computation*, 82:81–92, 1989.
13. B. Möller. Ideal stream algebra. In B. Möller and J. Tucker, editors, *Prospects for Hardware Foundations*, number 1546 in Lecture Notes in Computer Science, pages 69–116. Springer, 1998.
14. Object Management Group (OMG). *OMG Unified Modeling Language Specification, 3. UML Notation Guide, Part 9: Statechart Diagrams*, Mar. 2003.
15. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Object Technology Series. Addison-Wesley, 1998.
16. P. Scholz. Design of reactive systems and their distributed implementation with statecharts. PhD Thesis, TUM-I9821, Technische Universität München, Aug. 1998.
17. G. Ştefănescu. *Network Algebra*. Discrete Mathematics and Theoretical Computer Science. Springer, 2000.
18. R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.
19. A. Stümpel. *Stream Based Design of Distributed Systems through Refinement*. Logos Verlag Berlin, 2003.
20. P. Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40(5):80–91, May 1997.
21. G. Winskel and M. Nielsen. Models for concurrency. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Semantic Modelling*, volume 4 of *Handbook of Logic in Computer Science*, pages 1–148. Oxford University Press, 1995.

Generic Array Programming in SAC

Clemens Grelck[1] and Sven-Bodo Scholz[2]

[1] University of Lübeck, Germany
Institute of Software Technology and Programming Languages
grelck@isp.uni-luebeck.de
[2] University of Hertfordshire, United Kingdom
Department of Computer Science
s.scholz@herts.ac.uk

Abstract. SAC is a purely functional array processing language designed with compute-intensive numerical applications in mind. The declarative, generic style of programming in SAC is demonstrated by means of a small case study: 3-dimensional complex fast-Fourier transforms. The impact of abstraction on expressiveness, readability, and maintainability of code as well as on clarity of underlying mathematical concepts is discussed and compared with other approaches. The associated impact on runtime performance is quantified both in uniprocessor and in multiprocessor environments.

1 Introduction

Functional languages are generally considered well-suited for parallelization. Program execution is based on the principle of context-free substitution of expressions. Programs are free of side-effects and adhere to the Church-Rosser property. Any two subexpressions without data dependencies can be executed in parallel without any further analysis.

Classical domains of parallel computing like image processing or computational sciences are characterized by large arrays of numerical data [1]. Unfortunately, almost all functional languages focus on lists and trees, not on arrays. Notational support for multi-dimensional arrays is often rudimentary. Even worse, sequential runtime performance in terms of memory consumption and execution times fails to meet the requirements of numerical applications [2–4].

SAC (Single Assignment C) [5] is a purely functional array language. Its design aims at combining generic, high-level array processing with a runtime performance that is competitive with low-level machine-oriented programs written in C or FORTRAN. The core syntax of SAC is a subset of C with a strict, purely functional semantics based on context-free substitution of expressions. Nevertheless, the meaning of functional SAC code coincides with the state-based semantics of literally identical C code. This design is meant to facilitate conversion to SAC for programmers with a background in imperative languages.

The language kernel of SAC is extended by multi-dimensional, stateless arrays. In contrast to other array languages, SAC provides only a very small set of

W. Goerigk (Hrsg.): Programmiersprachen und Rechenkonzepte. Bericht Nr. 0410, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

built-in operations on arrays, mostly primitives to retrieve data pertaining to the structure and contents of arrays. All aggregate array operations are specified in SaC itself using a versatile and powerful array comprehension construct, named WITH-*loop*. WITH-loops allow code to abstract not only from concrete shapes of argument arrays, but even from concrete ranks (number of axes or number of dimensions) . Moreover, such rank-invariant specifications can be embedded within functions, which are applicable to arrays of any rank and shape.

By these means, most built-in operations known from FORTRAN-95 or from interpreted array languages like APL, J, or NIAL can be implemented in SaC itself without loss of generality [6]. SaC provides a comprehensive selection of array operations in the standard library. In contrast to array support which is hard-wired into the compiler, our library-based solution is easier to maintain, to extend, and to customize for varying requirements.

SaC propagates a programming methodology based on the principles of abstraction and composition. Like in APL, complex array operations and entire application programs are constructed by composition of simpler and more general operations in multiple layers of abstractions. Unlike APL, the most basic building blocks of this hierarchy of abstractions are implemented by WITH-loops, not built-in. Whenever a basic operation is found to be missing during program development, it can easily be added to the repertoire and reused in future projects.

Various case studies have shown that despite a generic style of programming SaC code is able to achieve runtime performance figures that are competitive with low-level, machine-oriented languages [7, 8, 5, 9]. We achieve this runtime behaviour by the consequent application of standard compiler optimizations in conjunction with a number of tailor-made array optimizations. They restructure code from a representation amenable to programmers and maintenance towards a representation suitable for efficient execution by machines [10, 5, 9, 11]. Fully compiler-directed parallelization techniques for shared memory architectures [12–14] further enhance performance. Utilization of a few additional processing resources often allow SaC programs to outperform even hand-optimized imperative codes without any additional programming effort.

The rest of the paper is organized as follows. Section 2 gives a short introduction to SaC, while Section 3 further elaborates on programming methodology. Section 4 applies the techniques to a well-known benchmark: 3-dimensional complex FFT. Section 5 provides a quantitative analysis, while Section 6 draws conclusions and outlines directions of future work.

2 SAC — Single Assignment C

Essentially, SaC is a functional subset of C extended by multi-dimensional stateless arrays as first class objects. Arrays in SaC are represented by two vectors. The *shape vector* specifies an array's rank and the number of elements along each axis. The *data vector* contains all elements of an array in row-major order. Array types include arrays of fixed shape, e.g. `int[3,7]`, arrays of fixed rank, e.g. `int[.,.]`, arrays of any rank, e.g. `int[+]`, and a most general type encom-

passing both arrays of any rank and scalars: `int[*]`. The hierarchy of array types induces a subtype relationship. SAC supports function overloading both with respect to different base types and with respect to the subtype relationship.

SAC provides a small set of built-in array operations, basically primitives to retrieve data pertaining to the structure and contents of arrays, e.g. an array's rank (`dim(`*array*`)`), its shape (`shape(`*array*`)`), or individual elements (*array*`[`*index-vector*`]`). Compound array operations are specified using WITH-loop expressions. As defined in Fig. 1, a WITH-loop basically consists of three parts: a *generator*, an *associated expression* and an *operation*.

WithLoopExpr ⇒ **with** *Generator* **:** *Expr Operation*
Generator ⇒ (*Expr Relop Identifier Relop Expr* ⌈ *Filter* ⌉)
Relop ⇒ `<=` | `<`
Operation ⇒ **genarray** (*Expr* ⌈ , *Expr* ⌉)
| **fold** (*FoldOp* , *Expr*)

Fig. 1. Syntax of with-loop expressions.

The operation determines the overall meaning of the WITH-loop. There are two variants: `genarray` and `fold`. With `genarray(` ***shp***`,` ***default***`)` the WITH-loop creates a new array. The expression ***shp*** must evaluate to an integer vector, which defines the shape of the array to be created. With `fold(` ***foldop***`,` ***neutral***`)` the WITH-loop specifies a reduction operation. In this case, ***foldop*** must be the name of an appropriate associative and commutative binary operation with neutral element specified by the expression ***neutral***.

The generator defines a set of index vectors along with an index variable representing elements of this set. Two expressions, which must evaluate to integer vectors of equal length, define lower and upper bounds of a rectangular index vector range. An optional filter may be used to further restrict generators to various kinds of grids; for simplification we omit this detail in the following.

For each element of the set of index vectors defined by the generator the associated expression is evaluated. Depending on the variant of WITH-loop, the resulting value is either used to initialize the corresponding element position of the array to be created (`genarray`), or it is given as an argument to the fold operation (`fold`). In the case of a `genarray`-WITH-loop, elements of the result array that are not covered by the generator are initialized by the (optional) default expression in the operation part. For example, the WITH-loop

```
with ([1,1] <= iv < [3,4]) : iv[0] + iv[1]
genarray( [3,5], 0)
```

yields the matrix $\begin{pmatrix} 0&0&0&0&0\\ 0&2&3&4&0\\ 0&3&4&5&0 \end{pmatrix}$ while the WITH-loop

```
with ([1,1] <= iv < [3,4]) : iv[0] + iv[1]
fold( +, 0)
```

evaluates to 21. More information on SAC is available at `www.sac-home.org`.

3 Programming methodology

As pointed out in the introduction, SaC propagates a programming methodology based on the principles of abstraction and composition. The usage of vectors in WITH-loop generators as well as in the selection of array elements along with the ability to define functions which are applicable to arrays of any rank and size allows us to implement generic compound array operations in SaC itself.

```
double[+] abs( double[+] a)
{
  res = with (. <= iv < shape(a)) : abs(a[iv])
        genarray( shape(a));
  return( res)
}

bool[+] (>=) ( double[+] a, double[+] b)
{
  res = with  (. <= iv <= .) : a[iv] >= b[iv]
        genarray( min( shape(a), shape(b)));
  return( res)
}

bool any( bool[+] a)
{
  res = with (0*shape(a) <= iv < shape(a)) : a[iv]
        fold( ||, false);
  return( res)
}
```

Fig. 2. Defining rank-invariant aggregate array operations in SaC.

Fig. 2 illustrates the principle of abstraction by rank-invariant definitions of three standard aggregate array operations. `abs` and `<=` extend the corresponding scalar functions to arrays of any rank and shape. The function `any` is a standard reduction operation, which yields `true` if any of the argument array elements is `true`, otherwise it yields `false`.

Some of the generators use the dot notation for lower or upper bounds. The dot represents the smallest or the largest legal index vector of the result array of a `genarray`-WITH-loop. The notation facilitates specification of frequent operations on all or on all inner elements of arrays.

In analogy to the examples in Fig. 2 most built-in operations known from other array languages can be implemented in SaC itself. The array module of the SaC standard library includes element-wise extensions of the usual arithmetic and relational operators, typical reduction operations like sum and product, various subarray selection facilities, as well as shift and rotate operations.

Basic array operations defined by WITH-loops lay the foundation to constructing more complex operations by means of composition, as illustrated in

```
bool cont( double[*] new, double[*] old, double eps)
{
  return( any( abs( new - old) >= eps))
}
```

Fig. 3. Defining array operations by composition.

Fig. 3. We define a generic convergence criterion for iterative algorithms of any kind purely by composition of basic array operations. Following this compositional style of programming, more and more complex operations and, eventually, entire application programs are built.

The strength of this generic rank-invariant programming style is the ability to specify array operations that are universally applicable to arrays of any shape, a property that is usually limited to built-in primitives in other languages.

4 Case study: NAS benchmark FT

In this section, we apply the generic programming techniques of SaC to a small but representative case study: 3-dimensional complex FFT. As part of the NAS benchmark suite [15] this numerical kernel has previously been used to assess the suitability of languages and compilers. Formal benchmarking rules and existing implementations in many languages ensure comparability of results. The NAS benchmark FT implements a solver for a class of partial differential equations by means of repeated 3-dimensional forward and inverse complex fast-Fourier transforms. They are implemented by consecutive collections of 1-dimensional FFTs on vectors along the three dimensions., i.e., an array of shape `[X,Y,Z]` is consecutively interpreted as a `ZY` matrix of vectors of length `X`, as a `ZX` matrix of vectors of length `Y`, and as a `XY` matrix of vectors of length `Z`.

```
complex[.,.,.] FFT( complex[.,.,.] a, complex[.] rofu)
{
  b = { [.,y,z] -> FFT( a[.,y,z], rofu) };
  c = { [x,.,z] -> FFT( b[x,.,z], rofu) };
  d = { [x,y,.] -> FFT( c[x,y,.], rofu) };

  return( d);
}
```

Fig. 4. SaC implementation of 3-dimensional FFT.

As shown in Fig. 4, the algorithm can be carried over into a SaC specification almost literally. The function `FFT` takes a 3-dimensional array of complex numbers (`complex[.,.,.]`) and consecutively applies 1-dimensional FFTs to all subvectors along the x-axis, the y-axis, and the z-axis. The SaC code takes advantage of the *axis control notation*. This notation facilitates specification of operations along one or multiple whole axes of argument arrays. Applications of this notation are transformed into WITH-loops in a pre-processing step. A

detailed introduction to both usage and compilation can be found in [16]. The additional parameter `rofu` provides a pre-computed vector of complex roots of unity, which is used for 1-dimensional FFTs.

```
complex[.] FFT(complex[.] v, complex[.] rofu)
{
  even      = condense(2, v);
  odd       = condense(2, rotate( [-1], v));
  rofu_even = condense(2, rofu);

  fft_even  = FFT1d( even, rofu_even);
  fft_odd   = FFT1d( odd,  rofu_even);

  left      = fft_even + fft_odd * rofu;
  right     = fft_even - fft_odd * rofu;

  return( left ++ right);
}

complex[2] FFT(complex[2] v, complex[1] rofu)
{
  return( [v[[0]] + v[[1]] , v[[0]] - v[[1]]]);
}
```

Fig. 5. SAC implementation of 1-dimensional FFT.

The overloaded function `FFT` on vectors of complex numbers (`complex[.]`) almost literally implements the Danielson-Lanczos algorithm [17]. It is based on the recursive decomposition of the argument vector `v` into elements at even and at odd index positions. The vector `even` can be created by means of the library function `condense(n,v)`, which selects every `n`-th element of `v`. The vector `odd` is generated in the same way after first rotating `v` by one index position to the left. `FFT` is then recursively applied to even and to odd elements, and the results are combined by a sequence of element-wise arithmetic operations on vectors of

```
typedef double[2] complex

complex (*) (complex a, complex b)
{
  return( [ a[0] * b[0] - a[1] * b[1],
            a[0] * b[1] + a[1] * b[0] ]);
}

complex[+] (*) (complex[+] a, complex[+] b)
{
  res = with (. <= iv <= .) : a[iv] * b[iv]
        genarray( min( shape(a), shape(b)));

  return( res);
}
```

Fig. 6. Complex numbers in SAC.

complex numbers and a final vector concatenation (++). A direct implementation of FFT on 2-element vectors (complex[2]) terminates the recursion.

Note that unlike FORTRAN neither the data type complex nor any of the operations used to define FFT are built-in in SAC. Fig.6 shows an excerpt from the complex numbers module of the SAC standard library.

```
subroutine  cffts1 ( is,d,x,xout,y)

include 'global.h'
integer is, d(3), logd(3)
double complex x(d(1),d(2),d(3))
double complex xout(d(1),d(2),d(3))
double complex y(fftblockpad, d(1), 2)
integer i, j, k, jj

do i = 1, 3
 logd(i) = ilog2(d(i))
end do

do k = 1, d(3)
 do jj = 0, d(2)-fftblock, fftblock
  do j = 1, fftblock
   do i = 1, d(1)
    y(j,i,1) = x(i,j+jj,k)
   enddo
  enddo

  call cfftz (is, logd(1),
              d(1), y, y(1,1,2))

  do j = 1, fftblock
   do i = 1, d(1)
    xout(i,j+jj,k) = y(j,i,1)
   enddo
  enddo
 enddo
enddo

return
end
```

```
subroutine fftz2 ( is,l,m,n,ny,ny1,u,x,y)

integer is,k,l,m,n,ny,ny1,n1,li,lj
integer lk,ku,i,j,i11,i12,i21,i22
double complex u,x,y,u1,x11,x21
dimension u(n), x(ny1,n), y(ny1,n)

n1 = n / 2
lk = 2 ** (l - 1)
li = 2 ** (m - l)
lj = 2 * lk
ku = li + 1

do i = 0, li - 1
 i11 = i * lk + 1
 i12 = i11 + n1
 i21 = i * lj + 1
 i22 = i21 + lk
 if (is .ge. 1) then
  u1 = u(ku+i)
 else
  u1 = dconjg (u(ku+i))
 endif
 do k = 0, lk - 1
  do j = 1, ny
   x11 = x(j,i11+k)
   x21 = x(j,i12+k)
   y(j,i21+k) = x11 + x21
   y(j,i22+k) = u1 * (x11 - x21)
  enddo
 enddo
enddo

return
end
```

Fig. 7. Excerpts from the FORTRAN-77 implementation of NAS-FT.

In order to help assessing the differences in programming style and abstraction, Fig. 7 shows excerpts from about 150 lines of corresponding FORTRAN-77 code. Three slightly different functions, i.e. cffts1, cffts2, and cffts3, intertwine the three transposition operations with a block-wise realization of a 1-dimensional FFT. The iteration is blocked along the middle dimension to improve cache performance. Extents of arrays are specified indirectly to allow reuse of the same set of buffers for all orientations of the problem. Function fftz2 is part of the 1-dimensional FFT. It must be noted that this excerpt represents high quality code, which is well organized and well structured. It was written by expert programmers in the field and has undergone several revisions. Everyday legacy FORTRAN-77 code is likely to be less "intuitive".

5 Experimental evaluation

This section investigates the runtime performance achieved by code compiled from the SAC specification of NAS-FT, as outlined in the previous section. It is

compared with that of the serial FORTRAN-77 reference implementation coming with the NAS benchmark suite 2.3[1], with a C implementation derived from the FORTRAN-77 code and extended by OPENMP directives[2] by Real World Computing Partnership (RWCP), and last but not least with the fastest HASKELL implementation proposed in [4]. All experiments were made on a 12-processor SUN Ultra Enterprise 4000 shared memory multiprocessor using SUN Workshop compilers.

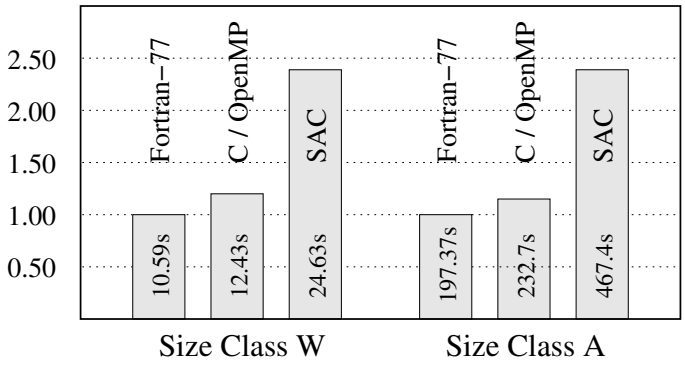

Fig. 8. Single processor performance of NAS-FT.

Fig. 8 shows sequential execution times for FORTRAN, C, and SAC. For both size classes investigated, FORTRAN-77 outperforms SAC by less than a factor of 2.4 while C outperforms SAC by less than a factor of 2.0. The performance of a few lines of highly generic SAC code is in reach of hand-optimized imperative implementations of the benchmark. The remaining performance gap must to a large extent be attributed to dynamic memory management overhead caused by the recursive decomposition of argument vectors when computing 1-dimensional FFTs. Unlike SAC, both imperative implementations use a static memory layout. HASKELL runtimes are omitted in Fig. 8 because with more than 27 minutes runtime for size class W it is more than 2 orders of magnitude slower than the other candidates. Furthermore, HASKELL fails altogether to compute size class A in a 32-bit address space. Therefore, we have excluded HASKELL from further experiments.

Fig. 9 shows the scalability achieved by the 3 candidates, i.e. parallel execution times divided by each candidate's best serial runtime. Whereas hardly any performance gain can be observed for automatic parallelization of the FORTRAN-77 code, SAC achieves speedups of up to 5.5 and up to 6.0 for size classes W and A, respectively. With these figures SAC even slightly outperforms OPENMP in terms of scalability.

Fig. 10 shows absolute runtimes using ten processors. Due to its superior sequential performance the C/OPENMP combination achieves the best abso-

[1] The source code is available at `http://www.nas.nasa.gov/Software/NPB/` .
[2] The source code is available at `http://phase.etl.go.jp/Omni/` .

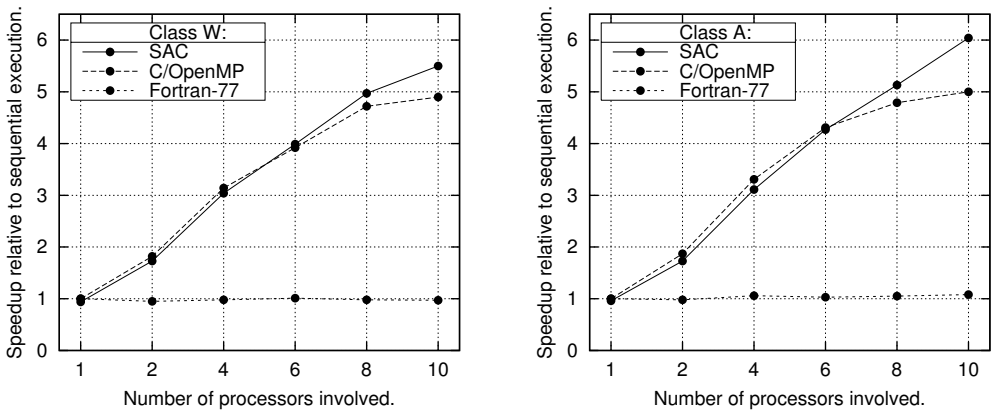

Fig. 9. Speedups achieved by multithreaded execution.

lute runtimes. However, this comes at the expense of 25 compiler directives for guiding parallelization. While parallelization of the SaC code is completely implicit like a compiler optimization, the resulting performance is still in reach of explicit approaches. It clearly outperforms automatic parallelization of the original Fortran-77 code.

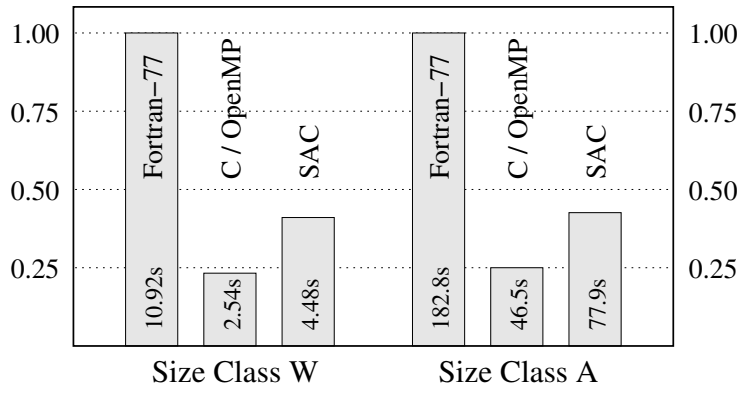

Fig. 10. 10-processor performance of NAS-FT.

6 Conclusions and future work

SaC aims at combining high-level, generic array programming with competitive runtime performance. The paper evaluates this approach based on the NAS benchmark FT. It is shown how 3-dimensional FFTs can be assembled by about 15 lines of SaC code as opposed to about 150 lines of fine-tuned Fortran-77 or C code. Due to its conciseness and high level of abstraction the SaC code clearly exhibits underlying mathematical algorithms, which are completely disguised by performance-related coding tricks in the case of Fortran-77 or C.

Development and maintenance of these codes require deep knowledge about computer architecture and corresponding optimization techniques, e.g. padding, tiling, buffering, or iteration ordering.

Nevertheless, the SaC runtime is within a factor of 2.4 of the Fortran-77 code and within a factor of 2.0 of the C code. In contrast, using the general-purpose functional language Haskell leads to a performance degradation of more than two orders of magnitude and prohibitive memory demands for non-trivial problem sizes. Furthermore, SaC by simple recompilation outperforms both low-level imperative implementations with only 4 processors of an SMP system. In contrast, only annotation with 25 OpenMP directives succeeded in exploiting multiple processors, whereas implicit parallelization of the Fortran-77 code failed to achieve any performance improvements.

Future work is basically twofold. First, various inefficiencies in the intermediate SaC code should be overcome by additional symbolic program transformations, which may allow us to further close the performance gap between SaC and low-level solutions. Second, we would like to extend the comparative study to other benchmark implementations, e.g. Mpi-based parallelization of Fortran-77 and C codes, a data parallel Hpf implementation, or a presumably faster Haskell implementation based on strict and unboxed arrays.

References

1. Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Frederickson, P., Lasinski, T., Schreiber, T., Simon, R., Venkatakrishnam, V., Weeratunga, S.: The NAS Parallel Benchmarks. International Journal of Supercomputer Applications **5** (1991) 63–73
2. Hartel, P., Langendoen, K.: Benchmarking Implementations of Lazy Functional Languages. In: Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA'93), Copenhagen, Denmark, ACM Press (1993) 341–349
3. Hartel, P., et al.: Benchmarking Implementations of Functional Languages with "Pseudoknot", a Float-Intensive Benchmark. Journal of Functional Programming **6** (1996)
4. Hammes, J., Sur, S., Böhm, W.: On the Effectiveness of Functional Language Features: NAS Benchmark FT. Journal of Functional Programming **7** (1997) 103–123
5. Scholz, S.B.: Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting. Journal of Functional Programming **13** (2003) 1005–1059
6. Grelck, C., Scholz, S.B.: Accelerating APL Programs with SAC. In Lefèvre, O., ed.: Proceedings of the International Conference on Array Processing Languages (APL'99), Scranton, Pennsylvania, USA. Volume 29 of APL Quote Quad. ACM Press (1999) 50–57
7. Grelck, C., Scholz, S.B.: HPF vs. SAC — A Case Study. In Bode, A., Ludwig, T., Karl, W., Wismüller, R., eds.: Proceedings of the 6th European Conference on Parallel Processing (Euro-Par'00), Munich, Germany. Volume 1900 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany (2000) 620–624

8. Grelck, C.: Implementing the NAS Benchmark MG in SAC. In Prasanna, V.K., Westrom, G., eds.: Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, Florida, USA, IEEE Computer Society Press (2002)
9. Grelck, C., Scholz, S.B.: SAC — From High-level Programming with Arrays to Efficient Parallel Execution. Parallel Processing Letters **13** (2003) 401–412
10. Scholz, S.B.: With-loop-folding in SAC — Condensing Consecutive Array Operations. In Clack, C., Davie, T., Hammond, K., eds.: Proceedings of the 9th International Workshop on Implementation of Functional Languages (IFL'97), St. Andrews, Scotland, UK, Selected Papers. Volume 1467 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany (1998) 72–92
11. Grelck, C., Scholz, S.B., Trojahner, K.: With-Loop Scalarization: Merging Nested Array Operations. In Trinder, P., Michaelson, G., eds.: Proceedings of the 15th International Workshop on Implementation of Functional Languages (IFL'03), Edinburgh, Scotland, UK, Revised Selected Papers. Volume 3145 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany (2004)
12. Grelck, C.: Shared Memory Multiprocessor Support for SAC. In Hammond, K., Davie, T., Clack, C., eds.: Proceedings of the 10th International Workshop on Implementation of Functional Languages (IFL'98), London, UK, Selected Papers. Volume 1595 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany (1999) 38–54
13. Grelck, C.: Implicit Shared Memory Multiprocessor Support for the Functional Programming Language SAC — Single Assignment C. PhD thesis, Institute of Computer Science and Applied Mathematics, University of Kiel, Germany (2001). Logos Verlag, Berlin, 2001.
14. Grelck, C.: A Multithreaded Compiler Backend for High-Level Array Programming. In Hamza, M., ed.: Proceedings of the 21st International Multi-Conference on Applied Informatics (AI'03), Part II: International Conference on Parallel and Distributed Computing and Networks (PDCN'03), Innsbruck, Austria, ACTA Press, Anaheim, California, USA (2003) 478–484
15. Bailey, D., Harris, T., Saphir, W., van der Wijngaart, R., Woo, A., Yarrow, M.: The NAS Parallel Benchmarks 2.0. NAS 95-020, NASA Ames Research Center, Moffet Field, California, USA (1995)
16. Grelck, C., Scholz, S.B.: Axis Control in SAC. In Peña, R., Arts, T., eds.: Proceedings of the 14th International Workshop on Implementation of Functional Languages (IFL'02), Madrid, Spain, Revised Selected Papers. Volume 2670 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany (2003) 182–198
17. Press, W., Teukolsky, S., Vetterling, W., Flannery, B.: Numerical Recipes in C. Cambridge University Press, Cambridge, UK (1993)

Von mathematischen über algorithmische zu physikalischen Strukturen

Hermann von Issendorff
Hauptstr. 40, 21745 Hemmoor
hviss@issendorff.de

Erweiterte Zusammenfassung. In der Vergangenheit wurde auf diesem Workshop schon mehrfach darüber vorgetragen, dass eine Programmiersprache, die Aktonalgebra, definiert werden kann, mit der sich nicht nur wie mit klassischen Programmiersprachen Datenverarbeitung beschreiben lässt, sondern auch die Struktur und das Layout der Hardware-Systeme, auf denen die Datenverarbeitung ablaufen soll. Dies führt unmittelbar auf die Frage, wie sinnvoll es ist, Software, Hardware und Layout von Rechnersystemen gemeinsam in einem einheitlichen Formalismus zu beschreiben. Die Antwort lässt sich in drei Schlagworten zusammenfassen: Man gewinnt an formaler Korrektheit, Automatisierbarkeit und Wirtschaftlichkeit. Zwischen der Ablaufbeschreibung, die mit Programmiersprachen geschieht, und den Schaltungsstrukturen, auf denen die Datenverarbeitung stattfindet, klafft eine semantische Lücke, die mit Compilern, Hardware-Entwurfssystemen und Layoutverfahren bisher nur inselartig abgedeckt wird. Aktonalgebra schliesst diese Lücken. Sie garantiert damit durch-gängige formale Korrektheit, ermöglicht maschinelle Bearbeitung und bietet eine Basis für die automatische Übertragung von Software in Chips.

Dieser Vortrag befasst sich mit der Frage, was mathematische Algebren, Programmiersprachen und Aktonalgebra verbindet und unterscheidet. Wie sich zeigt, beschreibt diese Reihenfolge einen Weg zunehmender Konkretisierung

Die Strukturen klassischer mathematischer Formalismen sind abstrakt, genauer, abstrakt von den Raumzeiteigenschaften der Physik. Die Zeichenfolgen, mit denen algebraische Ausdrücke beschrieben werden, sind, sieht man von der ihnen aufgeprägten Semantik ab, nichts anderes als physikalisch-konkrete lineare Strukturen. Setzt man zwei verschiedene algebraische Ausdrücke gleich, wie das in den Axiomen geschieht, dann verzichtet man auf die konkrete strukturelle Information. Die Wirkung einer solchen Abstraktion lässt sich besonders klar am Kommutativaxiom erkennen. Das Kommutativaxiom entspricht physikalisch der Gleichsetzung einer Struktur mit seinem Spiegelbild. Dabei geht die in der Struktur enthaltene Ordnungsinformation verloren, was bedeutet, dass nicht mehr zwischen links und rechts unterschieden werden kann, und bei zeitlich interpretierten Strukturen nicht mehr zwischen früher und später. Es ist dann keine Ordnung mehr festlegbar, in der die Ausdrücke ausgewertet werden sollen.

Konventionelle Programmiersprachen dagegen sind spezielle Algebren, die eine explizite oder implizite Auswertungsordnung haben, in denen also kein Kommutativaxiom gilt. Andererseits beschreiben sie nur die Verarbeitung von Daten, aber nicht die Strukturen, auf denen die Verarbeitung stattfindet. Sie können deshalb als raumabstrakte aber zeitkonkrete Algebren bezeichnet werden.

Aktonalgebra geht noch einen Schritt weiter: Sie ist raumzeitkonkret. Mit dieser Raumzeitsemantik lassen sich diskrete Systeme, d.h. Maschinen, strukturell und

W. Goerigk (Hrsg.): Programmiersprachen und Rechenkonzepte. Bericht Nr. 0410, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

operational vollständig beschreiben. Zu jeder Maschine gibt es genau eine aktonalgebraische Beschreibung und umgekehrt.

Aktonalgebra hat einerseits die gleiche Berechnungsmächtigkeit wie alle universelle Programmiersprachen, zusätzlich aber auch die Fähigkeit, die räumlichen Strukturen beschreiben zu können, auf der die Datenverarbeitung stattfindet. Abstrahiert man von der Raumstruktur, dann bleiben nur die Eigenschaften der universellen Programmiersprachen erhalten Abstrahiert man vom Zeitverhalten der als Aktonen bezeichneten Komponenten, dann wird Aktonalgebra zu einer Programmiersprache für räumliche Strukturen, darunter insbesondere multiplanare Layouts.

Die formale Beschreibbarkeit von DV-Strukturen wurde in der Vergangenheit vielfach untersucht. Die zu diesem Zweck entworfenen Algebren oder Programmiersprachen sind jedoch alle raumabstrakt, d.h. alle Konstrukte, die nur in einer räumlichen Struktur realisierbar sind, werden verkapselt und sind damit einer analytischen Behandlung entzogen. Die Network Algebra [1] z.B. beschreibt nur planare Strukturen und verwendet dazu verkapselte Verbindungselemente wie Mehrfach-Kreuzung, -Verzweigung, -Vereinigung, und -Schleife. Einen ähnlichen Ansatz macht Möller [4], der von einer rein funktionalen Beschreibung ausgeht, in der die räumlichen Eigenschaften der verschiedenen Hardware-Elemente in speziellen Moduln versteckt werden. Ruby [2] beschreibt die Beziehungen zwischen Digitalschaltungen, aber ohne Bezug auf die konkrete Struktur. Einzig CADIC [3] ist als Layout-Sprache entwickelt, die konkrete planare Strukturen beschreibt. CADIC besitzt aber keine Funktionalität.

Die heute allgemein verwendeten Hardware-Programmiersprachen, wie z.B. VHDL oder Verilog, beschreiben nur die Funktionen einer Schaltung, nicht aber ihre Struktur. Die funktionale Beschreibung ist zudem auf die Auflistung Boolescher Funktionen zwischen Speichertakten, d.h. auf die Registertransfer-Ebene beschränkt. Da für das Layout keine Strukturinformation zur Verfügung steht, muss eine geeignete Schaltung durch Vertauschen der Komponenten sowie deren Verbindungen gesucht werden. Dies Verfahren ist bei der heute üblichen grossen Menge der Komponenten aufwändig und liefert üblicherweise nur suboptimale Lösungen.

In [5] wurde bereits gezeigt, dass Aktonalgebra so allgemein und elementar ist, dass konventionelle Programmiersprachen und mathematische Algebren in ihr ausgedrückt werden können. Mit einfachen Konversionsregeln, die eine Raumstruktur per Default hinzufügen oder von ihr abstrahieren, lassen sich Programmiersprachen oder mathematische Algebren in Aktonalgebra konvertieren oder umgekehrt.

Literatur

1. Stefanescu, Gh.: Network Algebra. Theoretical Springer-Verlag (2000)
2. Jones, G., Sheeran, M.: Circuit Design in Ruby. In: Staunstrup, J. (ed.): Formal Methods of VLSI Design. North Holland (1990) 13-70
3. Kolla, R., Molitor, P., Osthof, H.G.: Einführung in den VLSI-Entwurf. Teubner-Verlag, Stuttgart (1989)
4. Möller, B.: Deductive Hardware Design: A Functional Approach. In: Möller B., Tucker J.V. (eds): Prospects of Hardware Foundations, LNCS, Vol. 1546. Springer-Verlag (1998)
5. von Issendorff, H.: Algebraic Description of Physical Systems. In: Moreno-Diaz R., Buchberger B., Freire Nistel J.L (eds.): Computer Aided Systems Theory - EUROCAST'01, LNCS, Vol. 2178. Springer-Verlag (2001)

Hardware/Software-Codesign mit der MicroCore-Architektur

Ulrich Hoffmann (uho@xlerb.de)

www.microcore.org

Zusammenfassung

MicroCore ist eine skalierbare, zwei Stack, Harvard Prozessor-Architektur für eingebettete Systeme, die einfach in handelsüblichen FPGAs realisierbar ist, dort als 32-Bit-Variante nur einen Bruchteil ($< \frac{1}{4}$) der verfügbaren Logikelemente belegt und so genügend Raum für applikationsspezifische Erweiterungen läßt.

MicroCore erlaubt, beim Entwurf eingebetteter Systeme eine Hardware/Software-Codesign Strategie einzusetzen: Für eine spezifische Anwendung lassen sich kritische Teilfunktionalitäten innerhalb eines Entwurfsspektrums realisieren, das von Implementierungen allein durch Programme bis hin zu Implementierungen vollständig durch Schaltwerke reicht. Dies fördert den Entwurf von möglichst einfachen und damit beherrschbaren Lösungen bei gleichzeitig geringem Energieverbrauch (Tsugio Makimoto, Sony: "Cleverness Driven Devices"). Im Gegensatz zu einer vollautomatischen Partitionierung der Realisierung in Hardware und Software-Bestandteile erfolgt die Partionierung hier über weite Strecken durch einen Ingenieur-Ansatz – durch bewusste intellektuelle Entwurfsentscheidungen innerhalb eines vielschichtigen Problemraums – der insbesondere vollständige Transparenz der Realisierung gewährleistet.

MicroCore ist auf unterschiedliche Weisen erweiterbar. Zunächst lassen sich applikations-spezifische Schaltungen einfach über einen bereits zur Architektur gehörigen IO-Bus ansprechen. Außerdem stehen sogenannte User-Instruktionen zur Verfügung, die wahlweise Unterprogrammaufrufe auf vordefinierte Programmspeicherpositionen durchführen oder aber applikations-spezifische Schaltungen ansprechen. Dies ermöglicht schrittweise, abgesicherte Übergänge von reinen Software-Implementierungen, über hardware-unterstützte Implementierungen, bis hin zu vollständigen Hardware-Implementierungen kritischer Teilfunktionalitäten des eingebetten Systems.

Der Vortrag erläutert die MicroCore-Architektur und führt anhand der ganzzahligen Multiplikation vor, wie dieser schrittweise, abgesicherte Übergang von Software zur Hardware-Implementierung der ganzzahligen Multiplikation vollzogen werden kann.

W. Goerigk (Hrsg.): Programmiersprachen und Rechenkonzepte. Bericht Nr. 0410, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

APPLE:
Advanced Procedural Programming Language Elements

Christian Heinlein

Dept. of Computer Structures, University of Ulm, Germany
heinlein@informatik.uni-ulm.de

Abstract. Today's programming languages have received a considerable degree of complexity, raising the question whether all the concepts provided are really necessary to solve typical programming problems. As an alternative to object-oriented and aspect-oriented languages, advanced procedural programming languages are suggested, which slightly extend the two basic concepts of classical procedural languages, i. e., data structures and procedures operating on them. By that means, it is possible to design programming languages which are much simpler to learn and use, while offering comparable expressiveness and flexibility.

1 Introduction

Today's programming languages, in particular aspect-oriented languages such as AspectJ [9], have received a considerable degree of complexity, making it both hard to learn their "vocabulary" (i. e., simply know all concepts and constructs offered by the language) and to "fluently speak" them (i. e., successfully apply these concepts and constructs in daily programming). In contrast, traditional procedural languages, such as Pascal or C, provided just two basic building blocks: *data structures* (records in particular) and *procedures* operating on them [1]. Modern procedural languages, such as Modula-2 or Ada, added the concept of *modules* to support encapsulation and information hiding [10]. In object-oriented languages such as Eiffel, Java, or C++, these separate and orthogonal entities have been combined into *classes* which offer *subtype polymorphism*, *inheritance* of data structures and procedures (which are usually called *methods* there), and *dynamic binding* of procedures as additional basic concepts.

Even though object-oriented languages support the construction of software that is usually more flexible, extensible, and reusable than traditional "procedural software," it soon turned out that many desirable properties are still missing. For example, *modular extensibility* (i. e., the ability to extend an existing system without modifying or recompiling its source code) is limited to adding new (sub)classes to a class hierarchy, while adding new operations (methods) to existing classes is impossible. Similarly, retroactively extending or modifying the behaviour of operations is infeasible. A great deal of research efforts have been expended in the past years to overcome these limitations by providing even more new concepts, e. g., open classes [3], or advice and inter-type member declarations in aspect-oriented languages [9], to name only a few.

Even though the set of these additional concepts is "sufficient" (in the sense that they indeed solve the problems encountered with object-oriented languages), the

W. Goerigk (Hrsg.): Programmiersprachen und Rechenkonzepte. Bericht Nr. 0410, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

question arises whether they are really "necessary" (in the sense that a smaller or simpler set of concepts would not be sufficient). Using AspectJ as an extreme example, this language provides eight more or less different kinds of "procedures," i. e., named blocks of executable code: static methods, instance methods and constructors defined in classes (as in the base language Java), plus instance methods and constructors defined as inter-type members in aspects, plus before, after, and around advice (still neglecting the distinction between "after returning," "after throwing," and general "after" advice).

Figure 1 illustrates this observation graphically: The road leading from procedural languages via object-oriented languages to "conceptually sufficient" aspect-oriented languages climbs up the "hill of complexity" by introducing more and more specialized language constructs in order to "patch" the original deficiencies of procedural and object-oriented languages. This hill of complexity is an undesired burden for language designers and implementors as well as for language users.

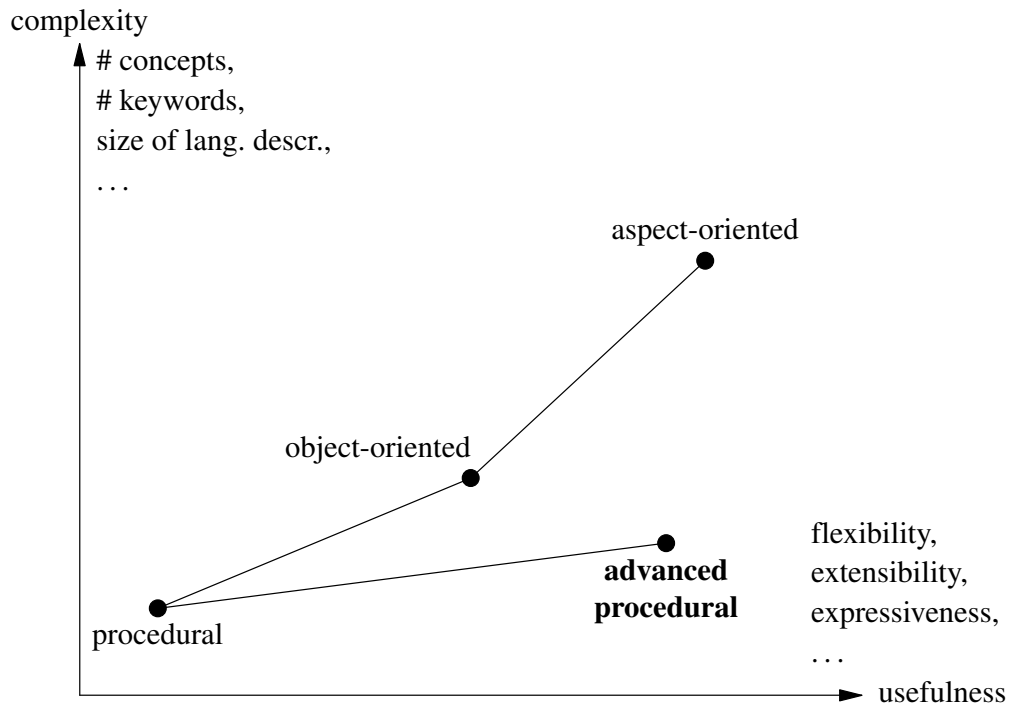

Figure 1: Hill of complexity

From a strictly conceptual point of view, this manifold of procedures is highly redundant: methods and constructors defined in classes are dispensable because they could always be defined in aspects; method and constructor bodies are dispensable because their code could always be defined as advice; before and after advice (the latter in its three variants) are dispensable because they are just special cases of around advice (calling `proceed` at the end resp. beginning of the advice, appropriately embedded in a `try`/`catch` block if necessary to distinguish the three variants of after ad-

vice). After these conceptual reductions, around advice – i. e., the possibility to freely override an existing "procedure," either with completely new code (that does not call `proceed`) or with code that augments the original code (by calling the latter via `proceed`) – remains as one of the *essential* (i. e., really necessary) procedure categories. (This goes in line with the statement that "dynamically scoped functions are the essence of AOP" [4].)

It turns out, however, that the potential for conceptual reductions is still not exhausted: By employing dynamic type tests (`instanceof` operator) in an around advice, a programmer is able to emulate the standard dynamic method dispatch provided by the base language Java (or any other dispatch strategy he likes) by executing the advice code only if the dynamic type of the current object is a particular subtype of its static type (or if some other arbitrary condition is satisfied) and simply call `proceed` otherwise. This means in consequence that the specialized built-in dispatch strategy for methods is dispensable from a purely conceptual point of view, thus removing the essential difference between statically and dynamically bound methods, i. e., between static and instance methods.

Similar considerations can be applied to data structures: data fields in classes are dispensable because they are just a special case of inter-type field declarations in aspects. Taken to the extreme, classes can always be declared with empty bodies, because their data fields, constructors, and methods can be declared more modularly and flexibly in aspects.

2 Suggestion

Given these observations, the basic suggestion of this paper is to go back to the starting point of procedural programming languages and extend them into a different direction in order to create *advanced procedural languages* which are significantly simpler than aspect-oriented languages while offering comparable expressiveness and flexibility (cf. Fig. 1).

In particular, replacing simple, statically bound procedures with arbitrarily overridable *dynamic procedures* (roughly comparable to around advice) covers (with some additional syntactic sugar which is not essential) the whole range of dynamic dispatch strategies usually found in object-oriented languages (single, multiple, and even predicate dispatch [2, 5]) plus the additional concept of advice (before, after, and around) introduced by aspect-oriented languages. Nevertheless, dynamic procedures remain a single, well-defined concept which is in no way entangled with data structures, class hierarchies, and the like and therefore is hardly more complex than traditional procedures.

Similarly, replacing simple record types having a fixed set of fields with modularly extensible *open types* and *attributes* (roughly comparable to empty classes extended by inter-type field declarations) covers classes and interfaces, field declarations in classes and aspects, multiple inheritance and subtype polymorphism, plus inter-type parent declarations and advice based on `get` and `set` pointcuts (since reading and writing attributes of open types is implicitly done via overridable dynamic proce-

dures). Again, open types constitute a single, well-defined concept which is little more complex than traditional record types.

Finally, preserving resp. (re-)introducing the *module* concept of modern procedural languages with clearly defined import/export interfaces and a strict separation of module definitions and implementations [12], provides perfect support for encapsulation and information hiding, even for applications where sophisticated concepts such as nested or friend classes are needed in today's languages [6, 11].

3 An Example of Open Types and Dynamic Procedures

This section presents a brief example of open types and dynamic procedures in "Advanced C." A little software library for the representation and evaluation of arithmetic expressions shall be developed.

We start by defining some open types with associated attributes.

```
// General expression.
type Expr;

// Constant expression.
type Const;

// Const is convertable to Expr, i. e. it is a subtype.
conv Const -> Expr;

// Value of constant expression.
attr val : Const -> int;

// Binary expression.
type Binary;
conv Binary -> Expr;     // Binary is a subtype of Expr, too.
attr op : Binary -> char;    // Operator and
attr left : Binary -> Expr;  // left and right
attr right : Binary -> Expr; // operand of binary expression.
```

Then, a dynamic procedure (or *global virtual function* in the nomenclature of C/C++) called `eval` is defined to compute the value of an expression.

```
// Evaluate constant expression.
// The static type of x is Expr, but this "branch" of eval
// is executed only if its dynamic type is Const.
virtual int eval (Expr x : Const) {
    return x@val; // @ is the attribute access operator
        // similar to the dot operator in other languages.
}

// Evaluate binary expression.
// This branch is executed if x's dynamic type is Binary.
virtual int eval (Expr x : Binary) {
```

```
    switch (x@op) {
    case '+': return eval(x@left) + eval(x@right);
    case '-': return eval(x@left) - eval(x@right);
    case '*': return eval(x@left) * eval(x@right);
    case '/': return eval(x@left) / eval(x@right);
    }
}
```

In a later stage of the development, we detect that we have forgotten to implement the remainder operator `%`. We fix this in a completely modular way (i. e., without the need to touch or recompile the above code) by adding another branch of `eval` overriding the previous one if the additional condition `x@op == '%'` is satisfied.

```
// Evaluate remainder expression.
// This branch is executed if x's dynamic type is Binary
// and the condition x@op == '%' holds.
virtual int eval (Expr x : Binary) if (x@op == '%') {
    return eval(x@left) % eval(x@right);
}
```

For a particular application of the library, we might want divisions by zero to return a special null value (represented, e. g., by the smallest available integer value) that propagates through all arithmetic operations (similar to the notion of "not a number" defined by IEEE 754 floating point arithmetics). This can be achieved, again in a completely modular way, by introducing the following additional branches of `eval`.

```
// Special null value.
const int null = INT_MIN;

// Catch divisions by zero.
virtual int eval (Expr x : Binary)
if (x@op == '/' || x@op == '%') {
    if (eval(x@right) == 0) return null;
    else return virtual(); // Call previous branch.
}

// Catch null-valued operands.
virtual int eval (Expr x : Binary) {
    if (eval(x@left) == null || eval(x@right) == null) {
        return null;
    }
    else {
        return virtual(); // Call previous branch.
    }
}
```

Note that the order in which the branches are defined is crucial in this example: Since the last branch – which will be tried first when the function is invoked – catches null-

valued operands, the second last branch will only be tried if both operands are not null and so does not need to repeat this test.

In contrast to normal object-oriented languages, where new classes can only be added as leaf nodes of the class hierarchy, new open types can also be inserted as inner nodes of the type hierarchy. For example, a new type `Atom` representing atomic expressions can be defined as a subtype of `Expr` and a supertype of `Const`:

```
// Atomic expression.
type Atom;

// Atom is a subtype of Expr.
conv Atom -> Expr;

// Const is a subtype of Atom.
conv Const -> Atom;
```

4 An Example from Operating Systems Development

Even though advanced procedural languages are intended to be general-purpose programming languages, their application to operating systems development might be particularly interesting since many of these systems are still implemented in traditional procedural languages (C in particular). Moving, e. g., from C to an "Advanced C" offering open types and dynamic functions should be much more smooth than shifting to an object-oriented or even aspect-oriented language, since the basic programming paradigm remains the same. Furthermore, by interpreting every standard C function as a dynamic function and every standard C struct as an open type with some initially associated attributes, it is possible to turn existing source code into flexibly extensible code at a glance, by simply recompiling it. With some system-dependent linker tricks it is even possible to turn standard library functions to dynamic functions without even recompiling them.

Operating systems, like software systems in general, usually evolve over time. Taking Unix and its derivatives as a typical example, this system started as a rather small and comprehensible system offering a few basic system calls which implemented a few fundamental concepts. Over the years and decades, it has grown into a large and complex system offering dozens of additional system calls implementing a large number of advanced concepts.

When using conventional programming languages, the introduction of each new concept typically requires modifications to numerous existing functions in addition to implementing new functions. Using open types and dynamic functions instead offers at least the chance to be able to implement new functionality in a truly *modular* way by grouping new data structures, necessary extensions to existing data structures, new functions, and necessary redefinitions of existing functions together in a single new unit of code.

To give a concrete example, the introduction of *mandatory file locking* into Unix required modifications to the implementation of several existing system calls (such as

`open`, `read`, and `write`) to make them respect *advisory locks* on a file (a concept that has been introduced earlier) as mandatory if the file's access permission bits contain an otherwise meaningless combination. Furthermore, this particular combination of access permissions has to be treated specially in other places of the system, e. g., by not performing the standard action of resetting the "set group ID on execution" bit when such a file is overwritten. By employing dynamic functions, modifications such as these can be implemented without touching or recompiling existing source code by simply overriding existing functions with new functions that perform additional tests before calling their previous implementation or signalling an error such as "lock violation" if necessary.

5 Conclusion

Advanced procedural programming languages have been suggested as an alternative direction to extend traditional procedural languages to make them more flexible and useful. In contrast to object-oriented and aspect-oriented languages, which combine the existing concepts of modules, data structures, and procedures into classes while at the same time introducing numerous additional concepts, advanced procedural languages retain these basic building blocks as orthogonal concepts which are only slightly extended to achieve the primary aim of modular extensibility.

Even though a first version of an "Advanced C" (that is actually being implemented as a language extension for C++ to get for free some of the advanced features of C++, such as templates and overloading of functions and operators) has been used successfully to implement some small to medium-sized programs (and there are also implementations available for dynamic procedures in Oberon and dynamic class methods in Java [7, 8]), it is too early yet to respectably report about experience and evaluation results. Of course, dynamic procedures are less efficient at run time than statically bound procedures because every explicit or implicit delegation of a call to the previous branch of the procedure is effectively another procedure call, at least when implemented straightforwardly without any optimizations. Furthermore, inlining of procedure calls becomes impossible if procedures can be freely redefined elsewhere. Nevertheless, the performance penalty encountered appears to be tolerable in practice if the concept is used reasonably.

It is often argued that the possibility to freely redefine procedures anywhere might quickly lead to incomprehensible code because this possibility might indeed be abused to completely change the behaviour of everything in a system. However, the limited practical experience gained so far suggests that the opposite is true, because when applied with care this possibility provides the unique ability to group related code together in a single place instead of needing to disperse it throughout a whole system. By that means, it is possible to develop and understand a system incrementally: Given that the basic functionality of the system is correct, it is possible to reason about its extensions separately in a modular way.

References

[1] A. V. Aho, J. E. Hopcroft: *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.

[2] C. Chambers, W. Chen: "Efficient Multiple and Predicate Dispatching." In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '1999)* (Denver, CO, November 1999). *ACM SIGPLAN Notices* 34 (10) October 1999, 238–255.

[3] C. Clifton, G. T. Leavens, C. Chambers, T. Millstein: "MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java." In: *Proc. 2000 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '00)* (Minneapolis, MN, October 2000). *ACM SIGPLAN Notices* 35 (10) October 2000, 130–145.

[4] P. Costanza: "Dynamically Scoped Functions as the Essence of AOP." *ACM SIGPLAN Notices* 38 (8) August 2003, 29–36.

[5] M. Ernst, C. Kaplan, C. Chambers: "Predicate Dispatching: A Unified Theory of Dispatch." In: E. Jul (ed.): *ECOOP'98 – Object-Oriented Programming* (12th European Conference; Brussels, Belgium, July 1998; Proceedings). Lecture Notes in Computer Science 1445, Springer-Verlag, Berlin, 1998, 186–211.

[6] J. Gosling, B. Joy, G. Steele: *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.

[7] C. Heinlein: *Vertical, Horizontal, and Behavioural Extensibility of Software Systems*. Nr. 2003-06, Ulmer Informatik-Berichte, Fakultät für Informatik, Universität Ulm, July 2003. http://www.informatik.uni-ulm.de/pw/berichte/

[8] C. Heinlein: "Dynamic Class Methods in Java." In: *Net.ObjectDays 2003. Tagungsband* (Erfurt, Germany, September 2003). tranSIT GmbH, Ilmenau, 2003, ISBN 3-9808628-2-8, 215–229.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold: "An Overview of AspectJ." In: J. Lindskov Knudsen (ed.): *ECOOP 2001 – Object-Oriented Programming* (15th European Conference; Budapest, Hungary, June 2001; Proceedings). Lecture Notes in Computer Science 2072, Springer-Verlag, Berlin, 2001, 327–353.

[10] D. L. Parnas: "On the Criteria to Be Used in Decomposing Systems into Modules." *Communications of the ACM* 15 (12) December 1972, 1053–1058.

[11] B. Stroustrup: *The C++ Programming Language* (Special Edition). Addison-Wesley, Reading, MA, 2000.

[12] N. Wirth: *Programming in Modula-2* (Fourth Edition). Springer-Verlag, Berlin, 1988.

What semantics fits with my aspects?

Ralf Lämmel [1,2]
[1] Vrije Universiteit, De Boelelaan 1081a, NL-1081 HV Amsterdam
[2] CWI, Kruislaan 413, NL-1098 SJ Amsterdam
mailto:ralf@cwi.nl
http://homepages.cwi.nl/~ralf/

Abstract: In this talk, we review the available semantics for aspect-oriented programming (AOP), and we connect this theme of recent research to pre-AOP age.

Most AOP semantics are operational or compiler-oriented in style, and they focus on idioms of AspectJ, which is the trend-setting, Java-based AOP language. A typical AOP semantics is based on a down-scaled Java, or perhaps on a simple functional language. The AOP-specific parts of the semantics are normally related to some form of aspect registry to keep track of intercepted join points and associated advice. Still these semantics differ with regard to the cunning peculiarities of describing pointcuts and managing the aspect registry. Furthermore, the semantics might or might not address static typing issues and other static guarantees. In addition to semantics, there are also foundational issues that are being studied, e.g., interpretations of AOP using process algebra (CSP) or the pi-calculus, and the static analysis of aspects for interference.

In fact, AOP foundations have been studied before the existence of AOP: think of structural and behavioural reflection as the most obvious example. Such imports from pre-AOP age also include continuation-passing style, wrapping, parallel programming, algorithmic debugging, and dynamic scoping.

Ongoing work on the semantics of AOP aims at "fluid" AOP, simpler semantical concepts, coverage of practical languages, modular reasoning, aspect composition, and semantics-preserving transformations of aspect-oriented programs.

Keywords: Aspect-Oriented Programming; Foundations; Formal Semantics

References

[1] R. Lämmel. A Semantical Approach to Method-Call Interception. In *Proc. of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, pages 41–55, Twente, The Netherlands, Apr. 2002. ACM Press.

[2] R. Lämmel. Adding Superimposition To a Language Semantics — Extended Abstract. In G. T. Leavens and C. Clifton, editors, *FOAL'03 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2002; Technical Report CS Dept., Iowa State Univ.*, Mar. 2003. 6 pages.

[3] R. Lämmel and K. Ostermann. Semantics of Fluid Lambdas (working title). Ongoing work, Apr. 2004.

[4] R. Lämmel and C. Stenzel. Semantics-Directed Implementation of Method-Call Interception. To appear in Journal IEE Proceedings Software; Special Issue on Unanticipated Software Evolution, 2004.

W. Goerigk (Hrsg.): Programmiersprachen und Rechenkonzepte. Bericht Nr. 0410, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

Demonstration of Factory - A Java Extension for Generative Programming

Christof Lutteroth

Institute of Computer Science
Freie Universität Berlin
Takustr.9, 14195 Berlin, Germany
lutterot@inf.fu-berlin.de

Abstract. Factory is an extension of Java which provides a template-based reflection mechanism for generative programming. Java classes can be parameterized by types, and the structure of the classes can be described dependent on these type parameters. Factory can address a wide range of applications and save programmers a lot of work. It is designed to integrate seamlessly with Java, to be intuitive for the user, extensible, and safe.

1 Introduction

Generative programming is about the idea to automate parts of the software development process. It is a paradigm that tries to bring the development of software onto a new level of abstraction and provide new means of reuse. Usually, there are programming tasks in software projects which are so regular that we can make the computer do them for us by programming a generator. A common example is a compiler, which generates the representation of a program in one language from its representation in another. Of course, we could do this translation by hand, but with the help of the compiler a lot of time is saved and we are able to focus on the aspects of our program on a level that is much more abstract than machine language. Programming languages, which are often implemented as compilers, can make a big difference to the programmer with the level of abstraction they provide.

There are many such tasks for which technologies of generative programming are used. For example, there are compiler generators, tools for the generation of database interfaces (e.g., [3]) and stub-generators (e.g. the Java *rmic* tool [12]). But it is not always an external tool that performs generation: Many programming languages have inbuilt features for generative programming, even though they are usually not seen from that point of view. A macro mechanism, for example, like the C preprocessor, is a generative feature, and also some features of object-oriented languages, like inheritance. In all such cases, the information given in the source-code is used to generate, according to a well-specified pattern, code with properties that are not expressed directly in the source.

Besides the widespread traditional mechanisms, like macros and inheritance, there are also more advanced ones. Nowadays, many programming languages

W. Goerigk (Hrsg.): Programmiersprachen und Rechenkonzepte. Bericht Nr. 0410, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

incorporate parametric polymorphism [4], also known as generic types, and introspective access to runtime objects (as, for example, provided by the Java reflection API [12]). Then, there exist more complex mechanisms, reflective architectures [6] that allow many aspects of a program to be changed at compile-time or even at runtime. For a general account on such technologies, see [7] or [5].

Factory [9] is a generative programming concept for problems for which parametric polymorphism without reflection, as known from C++ [11] and other languages [1], is not sufficient. The name "Factory" points out its generative capabilities on the level of metaprogramming and should not be confused with the factory design pattern. While the factory design-pattern refers to a class that creates objects, our factories are template-like entities that create classes. One could say that Factory takes the design pattern from the level of object generation to the metalevel of class generation.

Parametric polymorphism without reflection is used, e.g, for building type-safe containers, like lists or hash tables. But many important applications for generative programming require more powerful mechanisms. Examples that cannot be solved with traditional parametric polymorphism include those in which not only types are substituted but new signatures and code are created. A generator that yields EJB-conformant [10] wrappers for an arbitrary interface would be such an example. For each method of a given interface, the wrapper would have to generate a method with a different signature, dependent on the signature of the original method. Other examples include the generation of test suites for given classes, of a database interface, error handlers, stubs, etc. A generative approach to address these examples needs *introspection*, i.e., queries to the metamodel of the source-code that is dealt with, in order to explore all methods, as well as *intercession*, i.e., modifications of the metamodel, in order to create the new methods and their signatures (see also [6]).

Factory does support this kind of reflection, with a focus on *static reflection*, i.e., reflection done at compile-time. This is because in most practical examples, compile-time reflection seems to be sufficient. Runtime reflection is only needed in special cases, e.g., for hot deployment enabled containers in adaptive systems. Nevertheless, Factory is also capable of runtime reflection, but yet, compile-time reflection is easier to use.

Factory has been designed to provide a particularly strong notion of safety, *generator-type-safety*, by enabling certain analysis techniques of the Factory source-code representation at definition-time. Generator-type-safety guarantees the type-safety of all classes that can possibly be generated by a Factory generator. It can be verified with help of a type system [2]. Furthermore, Factory can determine if a generator terminates always.

2 The Factory Language

Factory has its own language that embeds a specialized XML-like syntax for compile-time generation into the standard syntax of Java 1.4. It uses the tem-

plate approach, so the programmer only needs to use Factory specific syntax whenever he wants to make use of its generative capabilities. Apart from that, programming is the same as in Java. However, the concept of Factory is not restricted to Java, and one day, it might be a good choice to integrate the compile-time part and the runtime part of the Factory syntax into a homogeneous, language independent abstract syntax.

The unit of generation and compilation is a *factory*, a file which contains exactly one class or interface definition (excluding inner interfaces/classes). It can be parameterized, use the Factory syntax of generative control constructs and generative terms, and invoke any number of other factories. But since the syntax is, apart from these Factory-specific extensions, that of a normal Java source file, all Java source files that contain only one class or interface (inner ones excluded) are valid factories.

2.1 Generator Variables

Generator variables are the variables used at generation-time, as opposed to the normal Java variables which are used at runtime. They are similar to normal Java variables, but they contain only object values. This is not a restriction, since the Factory system performs, when calling methods, casts to and from primitive types automatically. Furthermore, not all of these variables have an explicit type bound, but simply accept any object as value. Since the Factory generation system works with factory terms, which are functional, it is not possible to directly assign a value to a variable after it has been declared. There are several ways in which generator variables can be introduced into a factory: by declaring a Factory parameter with the `<param>` tag or by using the `<for>` or `<let>` constructs.

2.2 Factory Terms

Factory terms are a functional notation with which Java classes and other factories can be accessed in a safely restricted way. Usually, those terms are used to introspect the type parameters of the respective factory and to extract or construct the information that is needed for intercession.

It is one of the basic decisions in the design of Factory not to create a new metamodel for reflection but rather to integrate Factory seamlessly into the existing metaobject protocol of Java. This makes it a lot easier for people with a knowledge of the Java reflection API to use Factory. Furthermore, the integration with Java makes it easy to use and extend Factory with all the possibilities that Java offers. It is possible to access the standard classes as well as self-made ones, given that all the fields, methods and constructors that should be accessible are registered with the Factory system. Only classes that are considered safe, i.e., that terminate and have no harmful side effects, should be registered. The syntax of a Factory term is defined as follows:

term: (constant

| variable
| get
| application)

Internally, those terms work with objects, but any method that specifies parameters with primitive types can be used with objects of the corresponding class types. Also, if a method returns a value of a primitive type, it is internally converted into an object of the corresponding class type.

Constant literals can be created with constant terms that use the `<const>` tag. With this tag, objects for all the primitive Java types can be created, simply by using the respective literal standard notation. Also instances of metaclass `class` can be created by specifying the fully qualified class name.

Generator variables can be accessed with `<var>`:

variable: `<var>` IDENT `</var>`

Member variables of Java classes can be accessed with `<get>`. If the first tag in the body is another factory term, a member variable of the returned object is accessed; if it is a `<class>` tag, a static member variable of the named class is accessed.

get: `<get>`
(term
| `<class>` CLASS_IDENT `</class>`)
`<field>` VAR_IDENT `</field>`
`</get>`

Applications are done with an `<apply>` tag. If the first tag in the body is a term, a method is invoked on the object returned by that term. If it is a `<class>` tag followed by a `<method>` tag, the respective static method of the named class is invoked. If there is just the `<class>` tag, a constructor for the named class is invoked. If the `<factory>` is the first tag in the body of `<apply>`, a factory generator is applied, which returns the `Class` object for the generated class or interface. All applications may give arguments in the form of other terms in the `<args>` tag. If there is no argument, `<args>` can be left out.

application: `<apply>`
(term
`<method>` METHOD_IDENT `</method>`
| `<class>` CLASS_IDENT `</class>`
(`<method>` METHOD_IDENT `</method>`)$^{?}$
| `<factory>` FACTORY_IDENT `</factory>`)
(`<args>` (term)$^{+}$ `</args>`)$^{?}$
`</apply>`

2.3 Intercession with Factory Terms

Factory terms are placed at certain positions into standard Java source-code in order to perform intercession, i.e., to make the result of the term generate an element of the Java syntax. The terms are placeholders for Java syntax, which is filled in at generation-time.

Factory terms are only allowed at certain syntactical positions, and the position depends on the type of the term. In other words, if we want to generate a certain syntax element at a certain position, we have to use a term that evaluates to an object that models that syntax element correctly. The following table lists valid return types of Factory terms and positions where those terms can be used. In the second column, at the place of the term, we inserted the name of the class of which an object must be returned by the term.

Type	Possible Positions
`Class`	Instead of types: `Class` x; `Class` myMethod() ... int myMethod(`Class` x) ... x = (`Class`) y;

`String`	Instead of most identifiers:
	class `String` ...
	int `String`;
	x = `String` + 1;
	x = `String` (1);

`Package`	package `Package` ;
`Integer`	Instead of modifiers: `Integer` class C { ... } `Integer` int x; `Integer` int myMethod(...) { ... }
`Method`	Instead of method head: `Method` { ... }
`Class[]`	Instead of parameter or argument list: public int myMethod `Class[]` { ... } x = myMethod `Class[]`;
`Argument[]`	Instead of argument list: x = myMethod `Argument[]`;

2.4 Partial Evaluation

Factory terms can also be used in order to perform a simple partial evaluation. This optimization allows to do computations at generation-time and insert the result into the generated class.

```
1  class Calculator {
2    final double pi =
```

```
3       <literal>
4         <apply>
5           <class> myPackage.myClass </class>
6           <method> calcPi </method>
7         </apply>
8       </literal>;
9     ...
10  }
```

The `<literal>` tag can be used in Java expressions and must contain a term that evaluates to an object corresponding to a primitive Java type or `String` – those types, for which the Java syntax defines literals.

2.5 Control Constructs

In addition to terms, Factory provides control constructs: an `<if>`-tag for conditional generation, a `<for>`-tag for iterative generation, and a `<let>`-tag for assigning the value of a term to a new generator variable. Each of these constructs is available in two variants: one variant that can be used in place of a Java statement, e.g., in a method body, to generate statements, and one variant that can be used in place of a field to generate member variables and methods. Consequently, the nonterminal symbol element in the following rules can be either a statement or a field, depending on where the construct is placed.

The `<for>` allows to iterate over the elements of any array object. The array object must be given by the term after `<var>`. During generation, the fields or statements in the body are generated for each element in the array, and in each iteration, the respective element can be accessed in the generator variable declared in the `<var>` tag.

for:
`<for> <var>` IDENT `</var>` term
`<body>` (element)* `</body>`
`</for>`

The `<if>` allows conditional generation. The term after `<if>` must evaluate to an object of type `Boolean`, and if its value is true, the field(s)/statement(s) in the `<then>` tag are generated, otherwise the ones in the `<else>` tag. The `<else>` is optional.

if:
`<if>` term
`<then>` (element)* `</then>`
(`<else>` (element)* `</else>`)?
`</if>`

The let-construct allows to use a new generator variable in place of a term. The construct declares the variable and assigns it the value of the term, which can then be used in the `<body>`.

let:
<let> <var> IDENT </var> term
<body> (element)* </body>
</let>

The <let> can be seen as a special case of the <for> because it can be reduced to a loop that iterates over a single-element array.

3 Example for Compile-Time Reflection: Generating Setters

For an actual type parameter `Person`, the following factory `Setters` generates a class `PersonWithSetters` that extends class `Person`. `PersonWithSetters` overrides each public member variable of `Person` with a private member variable of the same type and name and declares a setter-method for each of it, similar to the convention of (non-enterprise-) JavaBeans.

This is more an academic example that demonstrates the reflexion capabilities of Factory. In real world applications, getter- and setter-methods are usually used in order to do something more than just reading or writing a single member variable, like notifying other objects when a value is changed.

```
1   <param>
2       <bound> java.lang.Object </bound>
3       <var> T </var>
4   </param>
5
6   public class
7       <apply>
8           <apply>
9               <var> T </var>
10              <method> getName </method>
11          </apply>
12          <method> concat </method>
13          <args> <const> "WithSetters" </const> </args>
14      </apply>
15  extends <var> T </var> {
16      <for> <var> I </var>
17      <apply> <var> T </var>
18          <method> getFields </method>
19      </apply> <body>
20          <let> <var> FT </var>
21          <apply> <var> I </var>
22              <method> getType </method>
23          </apply> <body>
24              <let> <var> FN </var>
25              <apply> <var> I </var>
```

```
26                  <method> getName </method>
27              </apply> <body>
28                  private <var> FT </var> <var> FN </var>;
29
30                  public void
31                  <apply>
32                      <const> "set" </const>
33                      <method> concat </method>
34                      <args> <var> FN </var> </args>
35                  </apply>
36                  (<var> FT </var> value) {
37                      this.<var> FN </var> = value;
38                  }
39          </body> </let> </body> </let>
40      </body> </for>
41  }
```

The following factory applies factory `Setters` to class `Person`, instantiates an object of the resulting class `PersonWithSetters` and uses its setter-method for its private member variable `plz`. If we uncomment line 12, which tries to access `plz` directly, we would not be able to compile it because a private member variable cannot be accessed outside of its class.

```
1   class SettersTest {
2       public static void main(String argv[]) {
3           <let> <var> T </var>
4           <apply>
5               <factory> Setters </factory>
6               <args> <const> Person </const> </args>
7           </apply>
8           <body>
9               <var> T </var> p = new <var> T </var>();
10              p.setPlz(1);
11
12              // p.plz++;
13          </body> </let>
14      }
15  }
```

4 Runtime Reflection

Since Factory is itself written in Java, it can easily be used from within other Java classes. This means that the generation process done by a factory can be invoked at runtime; and since Java supports dynamic class loading and reflective access to classes, the generated classes can be used straight away. This example demonstrates how Factory can be used for runtime reflection, as it is useful in

hot-deployment enabled environments. It would enable components, e.g., GUI components, to adapt dynamically to system changes, upgrades and extensions, providing a high degree of flexibility and tolerance. A type checker for generator-type-safety can statically ensure the dynamic safety of such adaptive components because it would assure that no adaption would bring the component into an erroneous state.

The following Java source-code snippet applies the factory `EditFrame` dynamically to the `Class` object in variable `t`: it creates an instance of factory `EditFrame` and uses its `facture()` method on `t` to generate a corresponding GUI component class `editFrame`, which is a GUI control for modifying the public member variables of instances of type `t`. The generated class is instantiated by means of the Java reflection API, and the `edit()` method called on that instance with an instance `o` of class `t` as argument.

```
1  Class editFrameClass =
2    new Factory("EditFrame")
3      .facture(t);
4  EditFrame myEditFrame =
5    (EditFrame) editFrameClass
6      .getConstructor(null)
7      .newInstance(null);
8  myEditFrame.edit(o);
```

5 Conclusion

We introduced the Factory language, outlined its syntax and semantics and described how it can be used to perform reflection and partial evaluation. Our idea was to formulate generators as templates. The template approach means that as much as possible of the desired output can be expressed directly. Often, this approach is already used for parametric polymorphism (e.g., in [11]), so it seemed straightforward to integrate into it further support for generative programming. In order to do advanced generation work, we implemented means to perform partial evaluation with the capability to introspect type parameters and to generate new signature elements and code. An important aim was to provide powerful reflection capabilities while still ensuring type-safety, i.e., a new, more general kind of safety that we call generator-type-safety.

We gave some examples in order to demonstrate the power and, above all, usefulness of Factory for real software development. Factory can be useful in the development of a wide range of applications. Like aspect oriented programming [8], it can address crosscutting concerns, i.e., functionality in a software system that is needed at different places, effectively by generating adapted subclasses and can be used dynamically in adaptive systems.

References

1. Boris Bokowski and Markus Dahm. Poor Man's Genericity for Java. In Serge Demeyer and Jan Bosch, editors, *Object-Oriented Technology, ECOOP'98 Workshop Reader, ECOOP'98 Workshops, Demos, and Posters, Brussels, Belgium, July 20-24, 1998, Proceedings*, volume 1543 of *Lecture Notes in Computer Science*, page 552. Springer, 1998.
2. Luca Cardelli. Type Systems. In *Handbook of Computer Science and Engineering*, chapter 103. CRC Press, 1997.
3. Oracle Corporation. Build Superior Java Applications with Oracle9iAS TopLink. Oracle Whitepaper, September 2002. http://otn.oracle.com/products/ias/toplink/TopLink_WP.pdf.
4. Benjamin C.Pierce. *Types and Programming Languages*. MIT Press, 2002.
5. K. Czarnecki and U. Eisenecker. *Generative Programming - Methods, Tools, and Applications*. Addison-Wesley, 2000.
6. Franois-Nicola Demers and Jacques Malenfant. Reflection in Logic, Functional and Object-oriented Programming: a Short Comparative Study, 1995.
7. Dirk Draheim, Christof Lutteroth, and Gerald Weber. An Analytical Comparison of Generative Programming Technologies. In *Proceedings of the 19. Workshop GI Working Group 2.1.4*. Technical Report at Christian-Albrechts-University of Kiel, November 2002.
8. Kiczales et al. An Overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 18–22. Budapest, Hungary, June 2001.
9. Christof Lutteroth. Factory, 2003. http://www.inf.fu-berlin.de/pj/factory/.
10. Sun Microsystems. Enterprise JavaBeans(TM) Specification 2.1 Proposed Final Draft 2, June 2003. http://java.sun.com/products/ejb/.
11. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, June 1997.
12. Sun Microsystems. *Java 2 SDK, Standard Edition - Documentation*, 2003. http://java.sun.com/j2se/1.4.2/docs/.

Constraint Solving for Generating Glass-Box Test Cases

Christoph Lembeck

Universität Münster

Abstract

Glass-box testing tries to cover paths through the tested code, based on a given criterion such as def-use chain coverage. When generating glass-box test cases manually, the user is likely to overlook some def-use chains. Moreover it is difficult to find suitable test cases which cause certain def-use chains to be passed. We have developed a tool which automatically generates a system of test cases for a piece of Java byte code, which ensures that all def-use chains are covered. The tool consists of a symbolic Java virtual machine (SJVM) and a system of dedicated constraint solvers. The SJVM uses the constraint solvers in order to determine which branches the symbolic execution needs to consider. A backtracking mechanism is applied in case that several branches remain feasible. Thus we have applied implementation techniques known from functional logic and constraint programming to handle the considered applications problems.

As already mentioned, the main difficulty of the approach above is checking, whether the collected constraints remain solvable, or if they are already contradicting each other. For this purpose we have implemented a dedicated constraint solver, which is integrated into our SJVM and is connected to the symbolic execution engine by a special constraint solver manager that administrates the gathering of new constraints and facilitates an incremental growth and solving of the constraints. The SJVM will be presented by Roger Müller and will not be explained in detail here. We will rather focus on the system of constraint solvers and the constraint solver manager, which acts as an interface between the symbolic execution engine of the SJVM and the different constraint solvers we have implemented to handle different kinds of constraints. Since the constraints produced by the execution engine arrive incrementally at the CSM, it has to store each of them for further calculations. As the backtracking mechanism of our tool also guarantees that the latest constraints added to the system are the first that will be removed again, the CSM needs a constraint stack to maintain them.

Moreover, the constraint solver manager analyzes the constraints and transforms them to some kind of normal form. Additionally, it selects the most appropriate constraint solver for each system of constraints and distributes each constraint to the corresponding constraint solver, in case that the overall system of constraints consisted of several independent subsystems. Therefor it is

W. Goerigk (Hrsg.): Programmiersprachen und Rechenkonzepte. Bericht Nr. 0410, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

Necessary to normalize the incomming constraints by removing fractions, modulo operations and the Java specific typecasts with the goal of getting easier manageable polynomial constraints.

As a result of the calculations of the constraint solvers the SJVM will be able to present the user a set of test cases consiting of both constraints and numerical values that lead to the specified test paths and the symbolical and numerical results representing the results of the program as they will be generated using a real virtual machine.

Issues in the Implementation of a Symbolic Java Virtual Machine for Test Case Generation

Roger A. Müller

Universität Münster

Abstract

Software testing is considered to be a more and more important part of the software development process. As manual testing is rather time-consuming, costly and most likely imprecise, developers increasingly rely on software testing tools. Software testing is commonly split into black- and glass-box testing. Black-box testing derives test cases from the specification of the user requirements, whereas glass-box testing is based solely on the code.

The test utility market has traditionally been dominated by tools that either manage test cases (i.e. regression test suites) or check existing test cases against a pre-defined coverage criterion, both glass- or black-box. If a coverage criterion is not met, it is usually up to the user to figure out which new test cases are required to gain a better coverage.

Our test tool generates test cases for glass-box testing automatically. Up to now, three ways to implement test case generation for structural criteria have been discovered. The first and simplest is the random generation of test cases. Obviously this technique is naive but easy to implement. The main drawback is that despite the high costs (every generated example has to be checked for validity and relevance) this method cannot guarantee the quality of its results. The dynamic approach actually executes the software. Test cases are discovered on the fly usually by local or global search (please refer to the related work section for more details). The static approach usually uses symbolic execution, which we will describe in a modified version in this paper.

Based on a user defined criterion a symbolic execution of the Java byte code is performed. Symbolic essentially means that the value of a variable is an expression depending on some input parameters (e.g. the parameters of the considered method) rather than a number. As a byproduct of the symbolic execution a system of equations is built, which describes the relation of the variables at the current instruction. If the symbolic execution reaches a branching instruction like ifgt, where a new constraint has to be added, a constraint solver is used in order to determine which branch to take. If two (or more) alternatives are still possible, the symbolic Java virtual machine (SJVM) tries them one by one using a backtracking mechanism similar to that of the implementation of logic and functional-logic languages like the Babel Abstract Machine (LBAM) or

W. Goerigk (Hrsg.): Programmiersprachen und Rechenkonzepte. Bericht Nr. 0410, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

the Warren Abstract Machine (WAM). At the end of the symbolic computation, one particular solution of the generated system of constraints will be determined. This solution represents a test case in the sense that the considered byte code has to produce the computed result, if executed with the appropriate values for the input parameter. Due to backtracking, alternative computation paths and the corresponding test cases will also be determined.

If the symbolic computation would closely follow the actual concrete computation, this would be too expensive and unnecessarily precise. Thus, in our approach the symbolic computation is guided by a user-specified coverage criterion. In the present work, we will focus on the well-known def-use-chain criterion

If the symbolic execution is performed for a given procedure or method the generated system of equations should be solvable for suitable input parameters. The solutions represent the input(s) required for a set of test cases that satisfy the given test criterion.

Electronic Notes in Theoretical Computer Science 80 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume80.html 18 pages

Formalizing Java's Two's-Complement Integral Type in Isabelle/HOL

Nicole Rauch [1]

Universität Kaiserslautern, Germany

Burkhart Wolff

Albert-Ludwigs-Universität Freiburg, Germany

Abstract

We present a formal model of the Java two's-complement integral arithmetics. The model directly formalizes the arithmetic operations as given in the Java Language Specification (JLS). The algebraic properties of these definitions are derived. Underspecifications and ambiguities in the JLS are pointed out and clarified. The theory is formally analyzed in Isabelle/HOL, that is, machine-checked proofs for the ring properties and divisor/remainder theorems etc. are provided. This work is suited to build the framework for machine-supported reasoning over arithmetic formulae in the context of Java source-code verification.

Key words: Java, Java Card, formal semantics, formal methods, tools, theorem proving, integer arithmetic.

1 Introduction

Admittedly, modelling numbers in a theorem prover is not really a "sexy subject" at first sight. Numbers are fundamental, well-studied and well-understood, and everyone is used to them since school-mathematics. Basic theories for the naturals, the integers and real numbers are available in all major theorem proving systems (e.g. [11,26,21]), so why care?

However, numbers as specified in a concrete processor or in a concrete programming language semantics are oriented towards an efficient implementation on a machine. They are finite datatypes and typically based on bit-fiddling definitions. Nevertheless, they often possess a surprisingly rich theory (ring properties, for example) that also comprises a number of highly non-standard and tricky laws with non-intuitive and subtle preconditions.

[1] Partially funded by IST VerifiCard (IST-2000-26328)

W. Goerigk (Hrsg.): Programmiersprachen und Rechenkonzepte. Bericht Nr. 0410, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

In the context of program verification tools (such as the B tool [2], KIV [3], LOOP [5], and Jive [20], which directly motivated this work), efficient numerical programs, e.g. square roots, trigonometric functions, fast Fourier transformation or efficient cryptographic algorithms represent a particular challenge. Fortunately, theorem proving technology has matured to a degree that the analysis of realistic machine number specifications for widely-used programming languages such as Java or C now is a routine task [13].
With respect to the formalization of integers, we distinguish two approaches:

(1) the *partial approach*: the arithmetic operations $+ - * / \%$ are only defined on an interval of (mathematical) integers, and left undefined whenever the result of the operation is outside the interval (c.f. [4], which is mainly geared towards this approach).
(2) the *wrap-around approach*: integers are defined on $[-2^{n-1} .. 2^{n-1} - 1]$, where in case of overflow the results of the arithmetic operations are mapped back into this interval through modulo calculations. These numbers can be equivalently realized by bitstrings of length n in the widely-used two's-complement representation system [10].

While in the formal methods community there is a widespread reluctance to integrate machine number models and therefore a tendency towards either (infinite) mathematical integers or the first approach ("either remain fully formal but focus on a smaller or simpler language [...]; or remain with the real language, but give up trying to achieve full formality." [23]), we strongly argue in favour of the second approach for the following reasons:

(1) In a wrap-around implementation, certain properties like "Maxint + 1 = Minint" hold. This has the consequence that crucial algebraic properties such as the associativity law "a + (b + c) = (a + b) + c" hold in the wrap-around approach, but not in the partial approach. The wrap-around approach is therefore more suited for automated reasoning.
(2) Simply using the mathematical operators on a subset of the mathematical integers does not handle surprising definitions of operators appropriately. E.g. in Java the result of an integer division is always rounded towards zero, and thus the corresponding modulo operation can return negative values. This is unusual in mathematics. Therefore, this naïve approach does not only disregard overflows and underflows but also disregards unconventionally defined operators.
(3) The Java type int is defined in terms of wrap-around in the Java Language Specification [12], so why should a programmer who strictly complies to it in an efficient program be punished by the lack of formal analysis tools?
(4) Many parts of the JLS have been analyzed formally — so why not the part concerning number representations? There are also definitions and claimed properties that should be checked; and there are also possible inconsistencies or underspecifications as in all other informal specifications.

As technical framework for our analysis we use Isabelle/HOL and the Isar proof documentation package, whose output is directly used throughout this paper (for lack of space, however, we will not present any proofs here. The complete documentation will be found in a forthcoming technical report). Isabelle [21] is a *generic* theorem prover, i.e. new object logics can be introduced by specifying their syntax and inference rules. Isabelle/HOL is an instance of Isabelle with Church's *higher-order logic* (HOL) [11], a classical logic with equality enriched by total polymorphic higher-order functions. In HOL, induction schemes can be expressed inside the logic, as well as (total) functional programs. Isabelle's methodology for safely building up large specifications is the decomposition of problems into *conservative extensions*. A conservative extension introduces new constants (by *constant definitions*) and types (by *type definitions*) only via axioms of a particular, machine-checked form; a proof that conservative extensions preserve consistency can be found in [11]. Among others, the HOL library provides conservative theories for the logical type *bool*, for the numbers such as *int* and for bitstrings *bin*.

1.1 Related Work

The formalization of IEEE floating point arithmetics has attracted the interest of researchers for some time [8,1], e.g. leading to concrete, industry strength verification technologies used in Intel's IA64 architecture [13].

In hardware verification, it is a routine task to verify two's complement number operations and their implementations on the gate level. Usually, variants of *binary decision diagrams* are used to represent functions over bit words canonically; thus, if a trusted function representation is identical to one generated from a highly complex implementation, the latter is verified. Meanwhile, addition, multiplication and restricted forms of operations involving division and remainder have been developed [15]. Unfortunately, it is known that one variant particularly suited for one operation is inherently intractable for another, etc. Moreover, general division and remainder functions have been proven to be intractable by word-level decision diagrams (WLDD) [24]. For all these reasons, the approach is unsuited to investigate the *theory* of two's complement numbers: for example, the theorem JavaInt-div-mod (see Section 4.2), which involves a mixture of all four operations, can only be proven up to a length of 9 bits, even with leading edge technology WLDD packages [2] .

Amazingly, *formalized theories* of the two's complement number have only been considered recently; i.e. Fox formalized 32-bit words and the ARM processor for HOL [9], and Bondyfalat developed a (quite rudimentary) bit words theory with division in the AOC project [6]. In the context of Java and the JLS, Jacobs [16] presented a fragment of the theory of integral types. This work (like ours) applies to Java Card as well since the models of the four smaller integral types (excluding long) of Java and Java Card are identical

[2] Thanks to Marc Herbstritt [14] to check this for us!

[25, § 2.2.3.1]. However, although our work is in spirit and scope very similar to [16], there are also significant differences:

- We use standard integer intervals as reference model for the arithmetic operations as well as two's-complement bitstrings for the bitshift and the bitwise AND, OR, XOR operations (which have not been covered by [16] anyway). Where required, we prove lemmas that show the isomorphy between these two representations.
- While [16] just presents the normal behavior of arithmetic expressions, we also cover the exceptional behavior for expressions like "x / 0" by adding a second theory layer with so-called strictness principles (see Sect. 6).
- [16] puts a strong emphasis on widening and narrowing operations which are required for the Java types short and byte. We currently only concentrate on the type int and therefore did not model widening and narrowing yet.

The Java Virtual Machine (JVM) [19] has been extensively modelled in the Project Bali [22]. However, the arithmetic operations in this JVM model are based on mathematical integers. Since our work is based on the same system, namely Isabelle2002, our model of a two's-complement integer datatype could replace the mathematical integers in this JVM theory.

1.2 Outline of this Paper

Section 2 introduces the core conservative definitions and the addition and multiplication, Section 3 presents the division and remainder theory and Section 4 gives the bitwise operations. Sections 2, 3 and 4 examine the *normal behavior*, while Section 5 describes the introduction of *exceptional behavior* into our arithmetic theory leading to operations which can deal with exceptions that may occur during calculations.

2 Formalizing the Normal Behavior Java Integers

The formalization of Java integers models the primitive Java type int as closely as possible. The programming language Java comes with a quite extensive language specification [12] which tries to be accurate and detailed. Nonetheless, there are several white spots in the Java integer specification which are pointed out in this paper. The language Java itself is platform-independent. The bit length of the data type int is fixed in a machine-independent way. This simplifies the modelling task. The JLS states about the integral types:

Java Language Specification [12], §4.2

> "The integral types are byte, short, int, and long, whose values are 8-bit, 16-bit, 32-bit and 64-bit signed two's-complement integers, respectively, and char, whose values are 16-bit unsigned integers representing Unicode characters. [...] The values of the integral types are integers in the following ranges: [...] For int, from −2147483648 to 2147483647, inclusive"

The Java int type and its range are formalized in Isabelle/HOL [21] this way:

constdefs

BitLength :: nat	BitLength ≡ 32
MinInt-int :: int	MinInt-int ≡ − (2 ^ (BitLength − 1))
MaxInt-int :: int	MaxInt-int ≡ 2^(BitLength − 1) − 1

Now we can introduce a new type for the desired integer range:
typedef JavaInt = {i. MinInt-int ≤ i ∧ i ≤ MaxInt-int}

This construct is the Isabelle/HOL shortcut for a type definition which defines the new type JavaInt isomorphic to the set of integers between MinInt-int and MaxInt-int. The isomorphism is established through the automatically provided (total) functions Abs-JavaInt :: int⇒JavaInt and Rep-JavaInt :: JavaInt⇒int and the two axioms y : {i. MinInt-int ≤ i ∧ i ≤ MaxInt-int} ⟹ Rep-JavaInt (Abs-JavaInt y) = y and Abs-JavaInt (Rep-JavaInt x) = x. Abs-JavaInt yields an arbitrary value if the argument is outside of the defining interval of JavaInt.
We define MinInt and MaxInt to be elements of the new type JavaInt:

constdefs

MinInt :: JavaInt	MinInt ≡ Abs-JavaInt MinInt-int
MaxInt :: JavaInt	MaxInt ≡ Abs-JavaInt MaxInt-int

In Java, calculations are only performed on values of the types int and long. Values of the three smaller integral types are widened first:

Java Language Specification [12], §4.2.2

> "If an integer operator other than a shift operator has at least one operand of type long, then the operation is carried out using 64-bit precision, and the result of the numerical operator is of type long. If the other operand is not long, it is first widened (§5.1.4) to type long by numeric promotion (§5.6). Otherwise, the operation is carried out using 32-bit precision, and the result of the numerical operator is of type int. If either operand is not an int, it is first widened to type int by numeric promotion. The built-in integer operators do not indicate overflow or underflow in any way."

This paper describes the formalization of the Java type int, therefore conversions between the different numerical types are not in the focus of this work. The integer types byte and short can easily be added as all calculations are performed on the type int anyways, so the only operations that need to be implemented are the widening to int, and the cast operations from int to byte and short, respectively. The Java type long can be added equally easily as our theory uses the bit length as a parameter, so one only need to change the definition of the bit length (see above) to gain a full theory for the Java type long, and again only the widening operations need to be added. Therefore, we only conentrate on the Java type int in the following.

Our model of Java int covers all side-effect-free operators. This excludes the operators ++ and −−, both in pre- and postfix notation. These operators return the value of the variable they are applied to while modifying the value stored in that variable independently from returning the value. We do not treat assignment of any kind either as it represents a side-effect as well. This

also disallows combined operators like `a += b` etc. which are a shortcut for `a = a + b`. This is in line with usual specification languages, e.g. JML [17], which also allows only side-effect-free operators in specifications. From a logical point of view, this makes sense as the specification is usually regarded as a set of predicates. In usual logics, predicates are side-effect-free. Thus, expressions with side-effects must be treated differently, either by special Hoare rules or by program transformation.

In our model, all operators are defined in Isabelle/HOL, and their properties as described in the JLS are proven, which ensures the validity of the definitions in our model. In the following, we quote the definitions from the JLS and present the Isabelle/HOL definitions and lemmas.

Our standard approach of defining the arithmetic operators on JavaInt is to convert the operands from JavaInt to Isabelle int, to apply the corresponding Isabelle int operation, and to convert the result back to JavaInt. The first conversion is performed by the representation function Rep-JavaInt (see above). The inverse conversion is performed by the function Int-to-JavaInt:

Int-to-JavaInt :: int ⇒ JavaInt
  Int-to-JavaInt (x::int) ≡ Abs-JavaInt(
    ((x + (−MinInt-int)) mod (2 ∗ (−MinInt-int))) + MinInt-int)

This function first adds (−MinInt) to the argument and then performs a modulo calculation by $2 * (-\text{MinInt})$ which maps the value into the interval $[0 \,..\, 2*(-\text{MinInt})-1]$ (which is equivalent to only regarding the lowest 32 bits), and finally subtracts the value that was initially added. This definition is identical to the function Abs-JavaInt on arguments which are already in JavaInt. Larger or smaller values are mapped to JavaInt values, extending the domain to int.

This standard approach is not followed for operators that are explicitly defined on the bit representation of the arguments. Our approach differs from the approach used by Jacobs [16] who exclusively uses bit representations for the integer representation as well as the operator definitions.

2.1 Unary Operators

This section gives the formalizations of the unary operators +, − and the bitwise complement operator ∼. The unary plus operator on int is equivalent to the identity function. This is not very challenging, thus we do not elaborate on this operator. In the JLS, the unary minus operator is defined in relation to the binary minus operator described below.

Java Language Specification [12], §15.15.4

"At run time, the value of the unary minus expression is the arithmetic negation of the promoted value of the operand. For integer values, negation is the same as subtraction from zero.(1)
[...] negation of the maximum negative int or long results in that same maximum negative number.(2)
[...] For all integer values x, −x equals (∼x)+1.(3)"

The unary minus operator is formalized as
uminus-def : − (x::JavaInt) ≡ Int-to-JavaInt (− Rep-JavaInt x)
We prove the three properties described in the JLS:

(1) **lemma** uminus-property: 0 − x = − (x::JavaInt)

(2) **lemma** uminus-MinInt: − MinInt = MinInt

(3) **lemma** uminus-bitcomplement: (∼ x) + 1 = − x

Note that the unary minus operator has two fixed points: 0 and MinInt. This leads some unexpected results, e.g. `Math.abs(MinInt)` = MinInt, a negative number. Also, many of the lemmas presented in this paper do not hold for MinInt and therefore exclude that value in their assumptions.
The bitwise complement operator is defined by unary and binary minus:

Java Language Specification [12], §15.15.5

"At run time, the value of the unary bitwise complement expression is the bitwise complement of the promoted value of the operand; note that, in all cases, ∼x equals (−x)−1."

This is formalized in Isabelle/HOL as follows:
constdefs
JavaInt-bitcomplement :: JavaInt ⇒ JavaInt
JavaInt-bitcomplement (x::JavaInt) ≡ (−x) − (1::JavaInt)
We use the notations ∼ and JavaInt-bitcomplement interchangeably.

3 Additive and Multiplicative Operators

3.1 Additive Operators

This section formalizes the binary + operator and the binary − operator.

Java Language Specification [12], §15.18.2

"The binary + operator performs addition when applied to two operands of numeric type, producing the sum of the operands. The binary - operator performs subtraction, producing the difference of two numeric operands.(1) [...]
Addition is a commutative operation if the operand expressions have no side effects. Integer addition is associative when the operands are all of the same type(2) [...]
If an integer addition overflows, then the result is the low-order bits of the mathematical sum as represented in some sufficiently large two's-complement format.(3)
If overflow occurs, then the sign of the result is not the same as the sign of the mathematical sum of the two operand values.(4)
For both integer and floating-point subtraction, it is always the case that a−b produces the same result as a+(−b).(5)"

(1) These two operators are defined in the standard way described above. We only give the definition of the binary + operator:
defs (**overloaded**)
add-def : x + y ≡ Int-to-JavaInt (Rep-JavaInt x + Rep-JavaInt y)

(2) This behavior is captured by the two lemmas

lemma JavaInt-add-commute: x + y = y + (x::JavaInt)
lemma JavaInt-add-assoc: x + y + z = x+(y+z::JavaInt)

(3) This requirement is already fulfilled by the definition.

(4) This specification can be expressed as
lemma JavaInt-add-overflow-sign :
(MaxInt-int < Rep-JavaInt a + Rep-JavaInt b) ⟶ (a + b < 0)
This is a good example of how inexact several parts of the Java Language Specification are. If indeed only overflow, i.e. regarding two operands whose sum is larger than MaxInt, is meant here, then why pose such a complicated question? "the sign of the mathematical sum of the two operand values" will always be positive in this case, so why talk about "the sign of the result is not the same"? It would be much clearer to state "the sign of the result is always negative". But what if the authors also wanted to describe underflow, i.e. negative overflow, which is sometimes also referred to as "overflow"? In §4.2.2 the JLS states "The built-in integer operators do not indicate overflow or underflow in any way." Thus, the term "underflow" is known to the authors and is used in the JLS. Why do they not use it in the context quoted above? This would also explain the complicated phrasing of the above formulation.
To clarify these matters, we add the lemma
lemma JavaInt-add-underflow-sign :
(Rep-JavaInt a + Rep-JavaInt b < MinInt-int) ⟶ (0 ≤ a + b)

(5) This has been formalized as
lemma diff-uminus: a − b = a + (−b::JavaInt)

3.2 Multiplication Operator

The multiplication operator is described and formalized as follows:

Java Language Specification [12], §15.17.1

> "The binary * operator performs multiplication, producing the product of its operands. Multiplication is a commutative operation if the operand expressions have no side effects. [...] integer multiplication is associative when the operands are all of the same type"

defs (**overloaded**)
times-def : x ∗ y ≡ Int-to-JavaInt (Rep-JavaInt x ∗ Rep-JavaInt y)
The commutativity and associativity are proven by the lemmas
lemma JavaInt-times-commute: (x::JavaInt) ∗ y = y ∗ x
lemma JavaInt-times-assoc: (x::JavaInt) ∗ y ∗ z = x ∗ (y ∗ z)

Java Language Specification [12], §15.17.1

> "If an integer multiplication overflows, then the result is the low-order bits of the mathematical product as represented in some sufficiently large two's-complement format. As a result, if overflow occurs, then the sign of the result may not be the same as the sign of the mathematical product of the two operand values."

This is again implicitly fulfilled by our standard modelling.

4 Division and Remainder Operators

4.1 Division Operator

In Java, the division operator produces the first surprise if compared to the mathematical definition of division, which is also used in Isabelle/HOL:

Java Language Specification [12], §15.17.2

"The binary / operator performs division, producing the quotient of its operands. [...] Integer division rounds toward 0. That is, the quotient produced for operands n and d that are integers after binary numeric promotion (§5.6.2) is an integer value q whose magnitude is as large as possible while satisfying $|d \times q| \leq |n|$; moreover, q is positive when $|n| \geq |d|$ and n and d have the same sign, but q is negative when $|n| \geq |d|$ and n and d have opposite signs.(1)
There is one special case that does not satisfy this rule: if the dividend is the negative integer of largest possible magnitude for its type, and the divisor is -1, then integer overflow occurs and the result is equal to the dividend.(2)
Despite the overflow, no exception is thrown in this case. On the other hand, if the value of the divisor in an integer division is 0, then an ArithmeticException is thrown.(3)"

This definition points out a major difference between the definition of division in Isabelle/HOL and Java. If the signs of dividend and divisor are different, the results differ by one because Java rounds towards 0 whereas Isabelle/HOL floors the result. Thus, the naïve approach of modelling Java integers by partialization of the corresponding operations of a theorem prover gives the wrong results in these cases.
We model the division by performing case distinctions:

defs (**overloaded**)
div-def : (x::JavaInt) div y ≡
if ((0 < x ∧ y < 0) ∨ (x < 0 ∧ 0 < y))
∧ ¬ (∃ z. Rep-JavaInt x = z ∗ Rep-JavaInt y) then
Int-to-JavaInt (Rep-JavaInt x div Rep-JavaInt y) + 1
else
Int-to-JavaInt(Rep-JavaInt x div Rep-JavaInt y)

The properties mentioned in the language report are formalized as follows:

(1) **lemma** quotient-sign-plus :
((abs d ≤ abs n) ∨ (n = MinInt)) ∧ (n ≠ MinInt ∨ d ≠ -1)
∧ (neg (Rep-JavaInt n) = neg (Rep-JavaInt d)) ∧ d ≠ 0
⟹ 0 < (n div d)
lemma quotient-sign-minus :
((abs d ≤ abs n) ∨ (n = MinInt)) ∧ (n ≠ MinInt ∨ d ≠ -1)
∧ (neg (Rep-JavaInt n) ≠ neg (Rep-JavaInt d)) ∧ d ≠ 0
⟹ (n div d) < 0
The predicate "neg" holds iff the value of its argument is less than zero. We have to treat the case n = MinInt separately because the abs function on JavaInt produces an unusable result for MinInt (see above).

Again, the phrasing in the JLS is quite imprecise as the "one special

case that does not satisfy this rule" refers to both of the lemmas above.

(2) **lemma** JavaInt-div-minusone : MinInt div -1 = MinInt

(3) is not modelled by the theory presented in this section because this theory does not introduce a bottom element for integers in order to treat exceptional cases. Our model returns 0 in this case due to the definition of the total function div in Isabelle/HOL. Exceptions are handled by the next theory layer (see Sect. 6) which adds a bottom element to JavaInt and lifts all operations in order to treat exceptions appropriately.

Again, the JLS is not very elaborate in (2) regarding the sign of the resulting value if the magnitude of the dividend is less than the magnitude of the divisor. It would have been clearer had they stated the result instead of letting the reader derive the result from the presented inequalities.

4.2 Remainder Operator

The remainder operator is closely related to the division operator. Thus, it does not conform to standard mathematical definitions either.

Java Language Specification [12], §15.17.3

"The binary % operator is said to yield the remainder of its operands from an implied division [...] The remainder operation for operands that are integers after binary numeric promotion (§5.6.2) produces a result value such that $(a/b) * b + (a\%b)$ is equal to a.(1)
This identity holds even in the special case that the dividend is the negative integer of largest possible magnitude for its type and the divisor is -1 (the remainder is 0).(2)
It follows from this rule that the result of the remainder operation can be negative only if the dividend is negative, and can be positive only if the dividend is positive;(3)
moreover, the magnitude of the result is always less than the magnitude of the divisor.(4)
If the value of the divisor for an integer remainder operator is 0, then an ArithmeticException is thrown.(5)

Examples:	5%3 produces 2	(note that 5/3 produces 1)
	5%(-3) produces 2	(note that 5/(-3) produces -1)
	(-5)%3 produces -2	(note that (-5)/3 produces -1)
	(-5)%(-3) produces -2	(note that (-5)/(-3) produces 1)(6)"

When formalizing the remainder operator, we have to keep in mind the formalization of the division operator and the required equality (1). Therefore, the remainder operator mod is formalized as follows:

```
mod-def : (x::JavaInt) mod y ≡
    if ((0 < x ∧ y < 0) ∨ (x < 0 ∧ 0 < y))
     ∧ ¬ (∃ z. Rep-JavaInt x = z * Rep-JavaInt y) then
       Int-to-JavaInt( Rep-JavaInt x mod Rep-JavaInt y ) − y
    else
       Int-to-JavaInt( Rep-JavaInt x mod Rep-JavaInt y )
```

The formulations in the JLS give rise to the following lemmas:

(1) **lemma** JavaInt-div-mod : ((a::Javalnt) div b) ∗ b + (a mod b) = a

(2) **lemma** MinInt-mod-minusone: MinInt mod −1 = 0
lemma MinInt-minusone-div-mod-eq :
(MinInt div −1) ∗ (−1) + (MinInt mod −1) = MinInt

(3) These phrases are not at all clear to us. We formalized them as follows, in the hope of meeting the intentions of the authors:
lemma neg-mod-sign : (a::Javalnt) < 0 ∧ b ≠ 0 ⟹ a mod b ≤ 0
lemma pos-mod-sign : 0 ≤ (a::Javalnt) ∧ b ≠ 0 ∧ b ≠ MinInt
⟹ 0 ≤ a mod b

(4) **lemma** JavaInt-mod-less : b ≠ 0 ∧ b ≠ MinInt
⟹ abs ((a::Javalnt) mod b) < abs b
It is not clear whether the "magnitude of the result" refers to the mathematical absolute value or to the Java method `Math.abs`. We decided to use the function abs on JavaInt which allows us to stay in the abstract model. This has the drawback that the lemma cannot be used for b = MinInt.

(5) See the discussion for div above.

(6) **lemma** div-mod-example1 : (5::Javalnt) mod 3 = 2 etc.

Again, it is not made explicit in the JLS what happens if the dividend equals 0.

Java is not the only language whose definitions of div and mod do not resemble the mathematical definitions. The languages Fortran, Pascal and Ada define division in the same way as Java, and Fortran's MOD and Ada's REM operators are modelled in the same way as Java's % operator. Goldberg [10, p. H-12] regrets this disagreement among programming languages and suggests the mathematical definition, some of whose advantages he points out.

5 Formalization With Bitstring Representation

5.1 Shift Operators

The shift operators are not properly described in the JLS (§15.19) either. It is especially unclear what happens if the right-hand-side operand of the shift operators is negative. Due to the space limitations of this paper, we have to refrain from presenting the full formalization of the shift operators here.

5.2 Relational Operators

As the relational operators (described in JLS §§15.20, 15.21) do not offer many surprises, we abstain from presenting their formalization here.

5.3 Integer Bitwise Operators &, ˆ, and |

This section formalizes the bitwise AND, OR, and exclusive OR operators.

Java Language Specification [12], §15.22, 15.22.1

"The bitwise operators [...] include the AND operator &, exclusive OR operator ˆ, and inclusive OR operator |.(1) [...]
Each operator is commutative if the operand expressions have no side effects. Each operator is associative.(2) [...]
For &, the result value is the bitwise AND of the operand values. For ˆ, the result value is the bitwise exclusive OR of the operand values. For |, the result value is the bitwise inclusive OR of the operand values. For example, the result of the expression 0xff00 & 0xf0f0 is 0xf000. The result of 0xff00 ˆ 0xf0f0 is 0x0ff0. The result of 0xff00 | 0xf0f0 is 0xfff0.(3)"

(1) These bitwise operators are formalized as follows:
constdefs
JavaInt-bitand :: [JavaInt,JavaInt] ⇒ JavaInt
x & y ≡ number-of (zip-bin (op &::[bool,bool]⇒bool)
(bin-of x) (bin-of y))
where bin-of transforms a JavaInt into its bitstring representation, zip-bin merges two bitstrings into one by applying a function (which is passed as the first argument) to each bit pair in turn, and number-of turns the resulting bitstring back into a JavaInt. The other two bit operators are defined accordingly.

(2) The commutativity and associativity of the three operators is proven by six lemmas, of which we present two here:
lemma bitand-commute: a & b = b & a
lemma bitand-assoc: (a & b) & c = a & (b & c)

(3) We verify the results of the examples by proving the three lemmas
lemma bitand-example : 65280 & 61680 = 61440
lemma bitxor-example : 65280 ˆ 61680 = 4080
lemma bitor-example : 65280 | 61680 = 65520
In these lemmas we transformed the hexadecimal values into decimal values because Isabelle is currently not able to read hex values.

5.4 Further Features of the Model

The model of Java integers presented above forms a ring. This could easily be proved by using Isabelle/HOL's Ring theory which only requires standard algebraic properties like associativity, commutativity and distributivity to be proven. The Ring theory makes dozens of ring theorems available for use in proofs. Our model also forms a linear ordering. To achieve this property, reflexivity, transitivity, antisymmetry and the fact that the ≤ operator imposes a total ordering had to be proven. This allows us to make use of Isabelle/HOL's linorder theory. We get a two's-complement representation by redefining (using our standard wrapper) the conversion function number-of-def

which is already provided for int. This representation is used for those operators that are defined bitwise. Altogether, the existing Isabelle theories make it relatively easy to achieve standard number-theoretic properties for types that are defined as a subset of the Isabelle/HOL integers.

5.5 Empirical Data: The Size of our Specification and Proofs

The formalization presented in the preceding sections consists of five theory files, the size of which is as follows:

Filename	Lines	Filename	Lines
JavaIntegersDef.thy	260	JavaIntegersAdd.thy	240
JavaIntegersTimes.thy	200	JavaIntegersRing.thy	500
JavaIntegersDiv.thy	1800	JavaIntegersBit.thy	350

It took about one week to specify the definitions and lemmas presented here and about eight to ten weeks to prove them, but the proof work was mainly performed by one of the authors (NR) who at the same time learned to use Isabelle, so an expert would be able to achieve these results much faster.

6 Formalizing the Exceptional Behavior Java Integers

The Java Language Specification introduces the concept of *exception* in expressions and statements of the language:

Java Language Specification [12], §11.3, §11.3.1

> "The control transfer that occurs when an exception is thrown causes abrupt completion of expressions (§15.6) and statements (§14.1) until a catch clause is encountered that can handle the exception [...]
> when the transfer of control takes place, all effects of the statements executed and expressions evaluated before the point from which the exception is thrown must appear to have taken place. No expressions, statements, or parts thereof that occur after the point from which the exception is thrown may appear to have been evaluated."

Thus, exceptions have two aspects in Java:

- they change the control flow of a program,
- they are a particular kind of side-effect (i.e. an exception object is created), and they prevent program parts from having side-effects.

While we deliberately neglect the latter aspect in our model (which can be handled in a Hoare Calculus on full Java, for example, when integrating our expression language into the statement language), we have to cope with the former aspect since it turns out to have dramatic consequences for the rules over Java expressions (these effects have not been made precise in the JLS).

So far, our normal behavior model is a completely denotational model; each expression is assigned a value by our semantic definitions. We maintain this

denotational view, with the consequence that we have to introduce *exceptional values* that are assigned to expressions that "may [not] appear to have been evaluated". In the language fragment we are considering, only one kind of exception may occur:

Java Language Specification [12], §4.2.2

"The only numeric operators that can throw an exception (§11) are the integer divide operator / (§15.17.2) and the integer remainder operator % (§15.17.3), which throw an ArithmeticException if the right-hand operand is zero."

In order to achieve a clean separation of concerns, we apply the technique developed in [7]. Conceptually, a theory morphism is used to convert a normal behavior model into a model enriched by exceptional behavior. Technically, the effect is achived by redefining all operators such as $+,-,*$ etc. using "semantical wrapper functions" and the normal behavior definitions given in the previous chapters. Two types of theory morphisms can be distinguished: One for a *one-exception world*, the other for a *multiple-exception world*. While the former is fully adequate for the arithmetic language fragment we are discussing throughout this paper, the latter is the basis for future extensions by e.g. array access constructs which may raise *out-of-bounds exceptions*. In the following, we therefore present the former in more detail and only outline the latter.

6.1 The One-Exception Theory Morphism

We begin with the introduction of a type constructor that disjointly adds to a type α a failure element such as $\bot$ (see e.g. [27], where the following construction is also called "lifting"). We declare a type class *bot* for all types containing a failure element $\bot$ and define as *semantical combinator*, i.e. as "wrapper function" of this theory morphism, the combinator *strictify* that turns a function into its *strict extension* wrt. the failure elements:

$$\begin{array}{rl} \text{strictify} & :: ((\alpha::\text{bot}) \Rightarrow (\beta::\text{bot})) \Rightarrow \alpha \Rightarrow \beta \\ \text{strictify f x} & \equiv \text{if x}=\bot \text{ then } \bot \text{ else f x} \end{array}$$

Moreover, we introduce the definedness predicate DEF :: α::bot $\Rightarrow$ bool by DEF x $\equiv$ (x $\neq \bot$). Now we introduce a concrete type constructor that lifts any type α into the type class bot:

$$\textbf{datatype} \quad \text{up}(\alpha) = \lfloor\_\rfloor\ \alpha \ \mid\ \bot$$

In the sequel, we write $t_\bot$ instead of up(t). We define the inverse to the constructor $\lfloor\_\rfloor$ as $\lceil\_\rceil$. Based on this infrastructure, we can now define the type JAVAINT that includes a failure element:

$$\textbf{types}\ \mathsf{JAVAINT} = \mathsf{JavaInt}_\bot$$

Furthermore, we can now define the operations on this enriched type; e.g. we convert the JavaInt unary minus operator into the related JAVAINT operator:

constdefs

uminus :: JAVAINT⇒JAVAINT

$$\text{uminus} \quad \equiv \text{strictify}(\lfloor\_\rfloor \circ \text{uminus} \circ \lceil\_\rceil)$$

As a canonical example for binary functions, we define the binary addition operator by (note that Isabelle supports overloading):

$$\textbf{op} + : [\mathsf{JAVAINT},\mathsf{JAVAINT}] \Rightarrow \mathsf{JAVAINT}$$
$$\textbf{op} + \equiv \text{strictify}(\lambda\, \mathrm{X}.\ \text{strictify}(\lambda\, \mathrm{Y}.\ \lfloor\lceil \mathrm{X}\rceil + \lceil \mathrm{Y}\rceil\rfloor))$$

All binary arithmetic operators that are strict extensions like $-$ or $*$ are constructed analogously; the equality and the logical operators like the strict logical AND & follow this scheme as well. For the division and modulo operators / and %, we add case distinctions whether the divisor is zero (yielding $\bot$). Java's non-strict logical AND && is defined in our framework by explicit case distinctions for $\bot$.

This adds new rules like $\mathrm{X} + \bot = \bot$ and $\bot + \mathrm{X} = \bot$. But what happens with the properties established for the normal behavior semantics? They can also be lifted, and this process can even be automated (see [7] for details). Thus, the commutativity and associativity laws for normal behavior, e.g. $(\mathrm{X} :: \mathsf{JavaInt}) + \mathrm{Y} = \mathrm{Y} + \mathrm{X}$, can be lifted to $(\mathrm{X} :: \mathsf{JAVAINT}) + \mathrm{Y} = \mathrm{Y} + \mathrm{X}$ by generic proof procedure establishing the case distinctions for failures. However, this works smoothly only if all variables occur on both sides of the equation; variables only occurring on one side have to be restricted to be defined. Consequently, the lifted version of the division theorem looks as follows:

$$[\![\ \text{DEF Y};\ \mathrm{Y} \neq 0\ ]\!] \Longrightarrow ((\mathrm{X} :: \mathsf{JAVAINT})\ /\ \mathrm{Y}) * \mathrm{Y} + (\mathrm{X}\ \%\ \mathrm{Y}) = \mathrm{X}$$

6.2 The Multiple-Exception Theory Morphism

The picture changes a little if the semantics of more general expressions are to be modelled, including e.g. array access which can possibly lead to out-of-bounds exceptions. Such a change of the model can be achieved by exchanging the theory morphism, leaving the normal behavior model unchanged.

It suffices to present the differences to the previous theory morphism here. Instead of the class bot we introduce the class exn requiring a family of undefined values $\bot_e$. The according type constructor is defined as:

$$\textbf{datatype} \quad \text{up}(\alpha) = \lfloor\_\rfloor\ \alpha \ \mid\ \bot \text{ exception}$$

and analogously to $\lceil\_\rceil$ we define exn-of$(\bot_e) = e$ as the inverse of the constructor $\bot$; exn-of is defined by an arbitrary but fixed HOL-value *arbitrary* for exn-of$(\lfloor_\rfloor) = \mathit{arbitrary}$. Definedness is $\mathrm{DEF}(x) = (\forall e.x \neq \bot_e)$.

The definition of operators is analogous to the previous section for the canonical cases; and the resulting lifting as well. Note, however, that the lifting of the commutativity laws fails and has to be restricted to the following:

$$[\![\ \text{DEF X} = \text{DEF Y} \wedge \text{exn-of X} = \text{exn-of Y}\ ]\!]$$
$$\Longrightarrow (\mathrm{X} :: \mathsf{JAVAINT}) + \mathrm{Y} = \mathrm{Y} + \mathrm{X}$$

These restrictions caused by the lifting reflect the fact that commutativity does not hold in a multi-exception world; if the left expression does not raise the same exception as the right, the expression order cannot be changed.

Hence, our proposed technique to use a theory morphism not only leads to a clear separation of concerns in the semantic description of Java, but also leads to the systematic introduction of the side-conditions of arithmetic laws in Java that are easily overlooked.

7 Conclusions and Future Work

In this paper we presented a formalization of Java's two's-complement integral types in Isabelle/HOL. Our formalization includes both normal and exceptional behavior. Such a formalization is a necessary prerequisite for the verification of efficient arithmetic Java programs such as encryption algorithms, in particular in tools like Jive [20] that generate verification conditions over arithmetic formulae from such programs.

Our formalization of the normal behavior is based on a direct analysis of the Java Language Specification [12] and led to the discovery of several underspecifications and ambiguities (see 3.1 (4), 4.1, 4.2, 5.1). These underspecifications are highly undesirable since even compliant Java compilers may interpret the same program differently, leading to unportable code. In the future, we strongly suggest to supplement informal language definitions by machine-checked specifications like the one we present in this paper as a part of the normative basis of a programming language.

We applied the technique of mechanized theory morphisms (developed in [7]) to our Java arithmetic model in order to achieve a clear separation of concerns between normal and exceptional behavior. Moreover, we showed that the concrete exceptional model can be exchanged — while controlling the exact side-conditions that are imposed by a concrete exceptional model. For the future, this leaves the option to use a lifting to the *exception state monad* [18] mapping the type JAVAINT to state $\Rightarrow$(JavaInt$_\perp$,state) in order to give semantics to expressions with side-effects like `i++ + i`.

Of course, more rules can be added to our theory in order to allow effective automatic computing of large (ground) expressions — this has not been in the focus of our interest so far. With respect to proof automation in JavaInt, it is an interesting question whether arithmetic decision procedures of most recent Isabelle versions (based on Cooper's algorithm for Presburger Arithmetic) can be used to decide analogous formulas based on machine arithmetic. While an *adoption* of these procedures to Java arithmetic seems impossible (this would require cancellation rules such as $(a \leq b) = (k \times a \leq k \times b)$ for nonnegative k which do not hold in Java), it is possible to retranslate JavaInt formulas to standard integer formulas; remainder sub-expressions can be replaced via $P(a \bmod b) = \exists m.\ 0 \leq m < a \wedge (a - m) \mid b \wedge P(m)$, such that finally a Presburger formula results. Since a translation leads to an exponential blow-

up in the number of quantifiers (a critical feature for Cooper's algorithm), it remains to be investigated to what extent this approach is feasible in practice.

References

[1] M. D. Aagaard and C.-J. H. Seger. The formal verification of a pipelined double-precision IEEE floating-point multiplier. In *Int. Conf. on Computer Aided Design*. IEEE Computer Society, 1995.

[2] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. CUP, 1996.

[3] M. Balser et al. Formal system development with KIV. In *Fundamental Approaches to Software Engineering*, LNCS 1783, 2000.

[4] B. Beckert and S. Schlager. Integer arithmetic in the specification and verification of Java programs. In *FM-TOOLS*, 2002.

[5] J. v. d. Berg and B. Jacobs. The LOOP compiler for Java and JML. In *TACAS01*, LNCS 2031, 2001.

[6] Dider Bondyfalat. Long integer division in Coq (algorithm divide and conquer). http://www-sop.inria.fr/lemme/Didier.Bondyfalat/DIV/.

[7] A. D. Brucker and B. Wolff. Using theory morphisms for implementing formal methods tools. In *Types for Proof and Programs*, LNCS, 2003.

[8] V. A. Carreño and P. S. Miner. Specification of the IEEE-854 floating-point standard in HOL and PVS. In *Higher Order Logic Theorem Proving and its Applications*, 1995.

[9] A. C. J. Fox. An algebraic framework for modelling and verifying microprocessors using HOL. TR 512, University of Cambridge, 2001.

[10] D. Goldberg. Computer arithmetic. In *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2002.

[11] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java^{TM} Language Specification – Second Edition*. Addison-Wesley, 2000.

[13] J. Harrison. A machine-checked theory of floating point arithmetic. In *Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS 1690, 1999.

[14] Marc Herbstritt. e-mail communication, May 2003. Chair of Computer Architecture, Uni Freiburg.

[15] S. Höreth and R. Drechsler. Formal verification of word-level specifications. In *IEEE Design, Automation and Test in Europe (DATE)*, 1999.

[16] Bart Jacobs. Java's integral types in PVS. *Submitted*, 2003.

[17] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*. Kluwer, 1999.

[18] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *POPL'95: Principles of Programming Languages*, 1995.

[19] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[20] J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In *TACAS00*, LNCS 276, 2000.

[21] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. LNCS 828. Springer, 1994.

[22] C. Pusch. Formalizing the Java Virtual Machine in Isabelle/HOL. Technical Report TUM-I9816, TU München, 1998.

[23] D. Sannella and A. Tarlecki. Algebraic methods for specification and formal development of programs. *ACM Computing Surveys*, 31(3es), 1999.

[24] C. Scholl, B. Becker, and T. Weis. On WLCDs and the complexity of word-level decision diagrams — a lower bound for division. *Formal Methods in System Design*, 20(3), 2002.

[25] Sun Microsystems, Inc. *Java Card^{TM} 2.1.1 Specifications – Release Notes*, 2000.

[26] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V7.3*, 2002.

[27] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

Functional Logic Programs with Dynamic Predicates*

– Extended Abstract –

Michael Hanus

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
mh@informatik.uni-kiel.de

Abstract. In this paper we propose a new concept to deal with dynamic predicates in functional logic programs. The definition of a dynamic predicate can change over time, i.e., one can add or remove facts that define this predicate. Our approach is easy to use and has a clear semantics that does not depend on the particular (demand-driven) evaluation strategy of the underlying implementation. In particular, the concept is not based on (unsafe) side effects so that the order of evaluation does not influence the computed results—an essential requirement in non-strict languages. Dynamic predicates can also be persistent so that their definitions are saved across invocations of programs. Thus, dynamic predicates are a lightweight alternative to the explicit use of external database systems. Moreover, they extend one of the classical application areas of logic programming to functional logic programs. We present the concept, its use and an implementation in a Prolog-based compiler.

1 Motivation and Related Work

Functional logic languages [10] aim to integrate the best features of functional and logic languages in order to provide a variety of programming concepts to the programmer. For instance, the concepts of demand-driven evaluation, higher-order functions, and polymorphic typing from functional programming can be combined with logic programming features like computing with partial information (logical variables), constraint solving, and non-deterministic search for solutions. This combination leads to optimal evaluation strategies [2] and new design patterns [4] that can be applied to provide better programming abstractions, e.g., for implementing graphical user interfaces [12] or programming dynamic web pages [13].

However, one of the traditional application areas of logic programming is not yet sufficiently covered in existing functional logic languages: the combination of declarative programs with persistent information, usually stored in relational databases, that can change over time. Logic programming provides a natural framework for this combination (e.g., see [7, 9]) since externally stored relations

* This research has been partially supported by the German Research Council (DFG) under grant Ha 2457/1-2.

W. Goerigk (Hrsg.): Programmiersprachen und Rechenkonzepte. Bericht Nr. 0410, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

can be considered as facts defining a predicate of a logic program. Thus, logic programming is an appropriate approach to deal with deductive databases or declarative knowledge management.

In this paper, we propose a similar concept for functional logic languages. Nevertheless, this is not just an adaptation of existing concepts to functional logic programming. We will show that the addition of advanced functional programming concepts, like the clean separation of imperative and declarative computations by the use of monads [24], provides a better handling of the dynamic behavior of database predicates, i.e., when we change the definition of such predicates by adding or removing facts. To motivate our approach, we shortly discuss the problems caused by traditional logic programming approaches to dynamic predicates.

The logic programming language Prolog allows to change the definition of predicates[1] by adding or deleting clauses using predefined predicates like `asserta` (adding a new *first* clause), `assertz` (adding a new *last* clause), or `retract` (deleting a matching clause). Problems occur if the use of these predicates is mixed with their update. For instance, if a new clause is added during the evaluation of a literal, it is not directly clear whether this new clause should be visible during backtracking, i.e., a new proof attempt for the same literal. This has been discussed in [18] where the so-called "logical view" of database updates is proposed. In the logical view, only the clauses that exist at the first proof attempt to a literal are used. Although this solves the problems related to backtracking, advanced evaluation strategies cause new problems.

It is well known that advanced control rules, like coroutining, provide a better control behavior w.r.t. the termination and efficiency of logic programs [21]. Although the completeness of SLD resolution w.r.t. any selection rule seems to justify such advanced control rules, it is not the case w.r.t. dynamic predicates. For instance, consider the Prolog program

```
ap(X) :- assertz(p(X)).

q :- ap(X), p(Y), X=1.
```

If there are no clauses for the dynamic predicate `p`, the proof of the literal `q` succeeds due to the left-to-right evaluation of the body of the clause for `q`. However, if we add the block declaration (in Sicstus-Prolog) "`:- block ap(-).`" to specify that `ap` should be executed only if its argument is not a free variable, then the proof of the literal `q` fails, because the clause for `p` has not been asserted when `p(Y)` should be proved.

This example indicates that care is needed when combining dynamic predicates and advanced control strategies. This is even more important in functional logic languages that are usually based on demand-driven (and concurrent) evaluation strategies where the exact order of evaluation is difficult to determine in advance [2, 11].

[1] In many Prolog systems, such predicates must be declared as "dynamic" in order to change their definitions dynamically.

Unfortunately, existing approaches to deal with dynamic predicates do not help here. For instance, Prolog and its extensions to persistent predicates stored in databases, like the Berkeley DB of Sicstus-Prolog or the persistence module of Ciao Prolog [6], suffer from the same problems. In the other hand, functional language bindings to databases do not offer the constraint solving and search facilities of logic languages. For instance, HaSQL[2] supports a simple connection to relational databases via I/O actions but provides no abstraction for computing queries (the programmer has to write SQL queries in plain text). This is improved in Haskell/DB [17] which allows to express queries through the use of specific operators. More complex information must be deduced by defining appropriate functions.

Other approaches to integrate functional logic programs with databases concentrate only on the semantical model for query languages. For instance, [1] proposes an integration of functional logic programming and relational databases by an extended data model and relational calculus. However, the problem of database updates is not considered and an implementation is not provided. Echahed and Serwe [8] propose a general framework for functional logic programming with processes and updates on clauses. Since they allow updates on arbitrary program clauses (rather than facts), it is unclear how to achieve an efficient implementation of this general model. Moreover, persistence is not covered in their approach.

Since real applications require the access and manipulation of persistent data, we propose a new model to deal with dynamic predicates in functional logic programs where we choose the declarative multi-paradigm language Curry [16] for concrete examples.[3] Although the basic idea is motivated by existing approaches (a dynamic predicate is considered as defined by a set of basic facts that can be externally stored), we propose a clear distinction between the accesses and updates to a dynamic predicate. In order to abstract from the concrete (demand-driven) evaluation strategy, we propose the use of time stamps to mark the lifetime of individual facts.

Dynamic predicates can also be persistent so that their definitions are saved across invocations of programs. Thus, our approach to dynamic predicates is a lightweight alternative to the explicit use of external database systems that can be easily applied. Nevertheless, one can also store dynamic predicates in an external database if the size of the dynamic predicate definitions becomes too large.

The next section contains a description of our proposal to integrate dynamic predicates into functional logic languages. Section 3 sketches a concrete implementation of this concept and Section 4 contains our conclusions. We assume familiarity with the concepts of functional logic programming [10] and Curry [11, 16].

[2] `http://members.tripod.com/~sproot/hasql.htm`

[3] Our proposal can be adapted to other modern functional logic languages that are based on the monadic I/O concept to integrate imperative and declarative computations in a clean manner, like Escher [19], Mercury [23], or Toy [20].

2 Dynamic Predicates

In this section we describe our proposal to dynamic predicates in functional logic programs and show its use by several examples.

2.1 General Concept

Since the definition of dynamic predicates is also intended to be stored persistently in files, we assume that dynamic predicates are defined by ground (i.e., variable-free) facts. However, in contrast to predicates that are explicitly defined in a program, the definition of a *dynamic* predicate is not provided in the program code but will be dynamically computed. Thus, dynamic predicates are similar to "external" functions whose code is not contained in the program but defined elsewhere. Therefore, the programmer has to specify in a program only the (monomorphic) type signature of a dynamic predicate (remember that Curry is strongly typed) and mark its name as "`dynamic`".

As a simple example, we want to define a dynamic predicate `prime` to store prime numbers whenever we compute them. Thus, we provide the following definition in our program:

```
prime :: Int -> Dynamic
prime dynamic
```

The predefined type "`Dynamic`" is abstract, i.e., there are no accessible data constructors of this type but a few predefined operations that act on objects of this type (see below). From a declarative point of view, `Dynamic` is similar to `Success` (the type of constraints), i.e., `prime` can be considered as a predicate. However, since the definition of dynamic predicates may change over time, the access to dynamic predicates is restricted in order to avoid the problems mentioned in Section 1. Thus, the use of the type `Dynamic` ensures that the specific access and update operations (see below) can be applied only to dynamic predicates. Furthermore, the keyword "`dynamic`" informs the compiler that the code for `prime` is not in the program but externally stored (similarly to the definition of external functions).

In order to avoid the problems related to mixing update and access to dynamic predicates, we put the corresponding operations into the I/O monad since this ensures a sequential evaluation order [24]. Thus, we provide the following predefined operations:

```
assert :: Dynamic -> IO ()

retract :: Dynamic -> IO Bool

getKnowledge :: IO (Dynamic -> Success)
```

`assert` adds a new fact about a dynamic predicate to the database where the *database* is considered as the set of all known facts for dynamic predicates. Actually, the database can also contain multiple entries (if the same fact is

repeatedly asserted) so that the database is a multi-set of facts. For the sake of simplicity, we ignore this detail and talk about sets in the following.

Since the facts defining dynamic predicates do not contain unbound variables (see above), `assert` is a rigid function, i.e., it suspends when the arguments (after evaluation to normal form) contain unbound variables. Similarly, `retract` is also rigid and removes a matching fact, if possible (this is indicated by the Boolean result value). For instance, the sequence of actions

```
assert (prime 1) >> assert (prime 2) >> retract (prime 1)
```

asserts the new fact `(prime 2)` to the database.

The action `getKnowledge` is intended to retrieve the set of facts stored in the database at the time when this action is executed. In order to provide access to the set of facts, `getKnowledge` returns a function of type "`Dynamic -> Success`" which can be applied to expressions of type "`Dynamic`", i.e., calls to dynamic predicates. For instance, the following sequence of actions (we use Haskell's "`do`" notation [22] in the following) asserts a new fact `(prime 2)` and retrieves its contents by unifying the logical variable `x` with the value `2`:[4]

```
do assert (prime 2)
   known <- getKnowledge
   doSolve (known (prime x))
```

Since there might be several facts that match a call to a dynamic predicate, we have to encapsulate the possible non-determinism occurring in a logic computation. This can be done in Curry by the primitive action to encapsulate the search for all solutions to a goal [5, 15]:

```
getAllSolutions :: (a -> Success) -> IO [a]
```

`getAllSolutions` takes a constraint abstraction and returns the list of all solutions, i.e., all values for the argument of the abstraction such that the constraint is satisfiable.[5] For instance, the evaluation of

```
getAllSolutions (\x -> known (prime x))
```

returns the list of all values for `x` such that `known (prime x)` is satisfied. Thus, we can define a function `printKnownPrimes` that prints the list of all known prime numbers as follows:

```
printKnownPrimes = do
   known <- getKnowledge
   sols <- getAllSolutions (\x -> known (prime x))
   print sols
```

[4] The action `doSolve` is defined as "`doSolve c | c = done`" and can be used to embed constraint solving into the I/O monad.

[5] `getAllSolutions` is an I/O action since the order of the result list might vary from time to time due to the order of non-deterministic evaluations.

Note that we can use all logic programming techniques also for dynamic predicates: we just have to pass the result of `getKnowledge` (i.e., the variable `known` above) into the clauses defining the deductive part of the database program and wrap all calls to a dynamic predicate with this result variable. For instance, we can print all prime pairs by the following definitions:

```
primePair known (x,y) =
   known (prime x) & known (prime y) & x+2 =:= y

printPrimePairs = do
   known <- getKnowledge
   sols <- getAllSolutions (\p -> primePair known p)
   print sols
```

The constraint `primePair` specifies the property of being a prime pair w.r.t. the knowledge `known`, and the action `printPrimePairs` prints all currently known prime pairs.

Our concept provides a clean separation between database updates and accesses. Since we get the knowledge at a particular point of time, we can access all facts independent on the order of evaluation. Actually, the order is difficult to determine due to the demand-driven evaluation strategy. For instance, consider the following sequence of actions:

```
do assert (prime 2)
   known1 <- getKnowledge
   assert (prime 3)
   assert (prime 5)
   known2 <- getKnowledge
   sols2 <- getAllSolutions (\x -> known2 (prime x))
   sols1 <- getAllSolutions (\x -> known1 (prime x))
   return (sols1,sols2)
```

Executing this code with the empty database, the pair of lists `([2],[2,3,5])` is returned. Although the concrete computation of all solutions is performed later than they are conceptually accessed (by `getKnowledge`) in the program text, we get the right facts (in contrast to Prolog with coroutining, see Section 1). Therefore, `getKnowledge` conceptually copies the current database for later access. However, since an actual copy of the database can be quite large, this is implemented by the use of time stamps (see Section 3).

2.2 Persistent Dynamic Predicates

One of the key features of our proposal is the easy handling of persistent data. The facts about dynamic predicates are usually stored in main memory which supports fast access. However, in most applications it is necessary to store the data also persistently so that the actual definitions of dynamic predicates survive different executions (or crashes) of the program. One approach is to store the facts in relational databases (which is non-trivial since we allow arbitrary term

structures as arguments). Another alternative is to store them in files (e.g., in XML format). In both cases the programmer has to consider the right format and access routines for each application. Our approach is much simpler (and often also more efficient if the size of the dynamic data is not extremely large): it is only necessary to declare the predicate as "persistent". For instance, if we want to store our knowledge about primes persistently, we define the predicate `prime` as follows:

```
prime :: Int -> Dynamic
prime persistent "file:prime_infos"
```

Here, `prime_infos` is the name of a directory where the run-time system automatically puts all files containing information about the dynamic predicate `prime`.[6] Apart from changing the `dynamic` declaration into a `persistent` declaration, nothing else needs to be changed in our program. Thus, the same actions like `assert`, `retract`, or `getKnowledge` can be used to change or access the persistent facts of `prime`. Nevertheless, the persistent declaration has important consequences:

- All facts and their changes are persistently stored, i.e., after a termination (or crash) and restart of the program, all facts are automatically recovered.
- Changes to dynamic predicates are immediately written into a log file so that they can be recovered.
- `getKnowledge` gets always the current knowledge persistently stored, i.e., if other processes also change the facts of the same predicate, it becomes immediately visible with the next call to `getKnowledge`.
- In order to avoid conflicts between concurrent processes working on the same dynamic predicates, there is also a transaction concept (which is not described in this extended abstract).

Note that the easy and clean addition of persistency was made possible due to our concept to separate the update and access to dynamic predicates. Since updates are put into the I/O monad, there are obvious points where changes must be logged. On the other hand, the `getKnowledge` action needs only a (usually short) synchronization with the external data and then the knowledge can be used with the efficiency of the internal program execution.

3 Implementation

In order to test our concept and to provide a reasonable implementation, we have implemented it in the PAKCS implementation of Curry [14]. The system compiles Curry programs into Prolog by transforming pattern matching into

[6] The prefix "`file:`" instructs the compiler to use a file-based implementation of persistent predicates. For future work, it is planned also to use relational databases to store persistent facts so that this prefix is used to distinguish the different access methods.

predicates and exploiting coroutining for the implementation of the concurrency features of Curry [3]. Due to the use of Prolog as the back-end language, the implementation of our concept is not very difficult. Therefore, we highlight only a few aspects of this implementation.

First of all, the compiler of PAKCS has to be adapted since the code for dynamic predicates must be different from other functions. Thus, the compiler translates a declaration of a dynamic predicate into specific code so that the run-time evaluation of a call to a dynamic predicate yields a data structure containing information about the actual arguments and the name of the external database (in case of persistent predicates). In this implementation, we have not used a relational database for storing the facts since this is not necessary for the size of the dynamic data (in our applications only a few megabytes). Instead, all facts are stored in main memory and in files in case of persistent predicates. First, we describe the implementation of non-persistent predicates.

Each `assert` and `retract` action is implemented via Prolog's `assert` and `retract`. However, as additional arguments we use time stamps to store the lifetime (birth and death) of all facts in order to implement the visibility of facts for the `getKnowledge` action (similarly to [18]). Thus, there is a global clock ("update counter") in the program that is incremented for each `assert` and `retract`. If a fact is asserted, it gets the actual time as birth time and ∞ as the death time. If a fact is retracted, it is not retracted in memory but only the death time is set to the actual time since there might be some unevaluated expression for which this fact is still visible. `getKnowledge` is implemented by returning a predefined function that keeps the current time as an argument. If this function is applied to some dynamic predicate, it unifies the predicate with all facts and, in case of a successful unification, it checks whether the time of the `getKnowledge` call is in the birth/death interval of this fact.

Persistent predicates are similarly implemented, i.e., all known facts are always kept in main memory. However, each update to a persistent predicate is written into a log file. Furthermore, all facts of this predicate are stored in a file in Prolog format. This file is only read and updated in the first call to `getKnowledge` or in subsequent calls if another concurrent process has changed the persistent data. In this case, the following operations are performed:

1. The previous database file with all Prolog facts is read.
2. All changes from the log file are replayed, i.e., executed.
3. A new version of the database file is written.
4. The log file is cleared.

In order to avoid problems in case of program crashes during this critical period, the initialization phase is made exclusive to one process via operating system locks and backup files are written.

4 Conclusions

We have proposed a new approach to deal with dynamic predicates in functional logic programs. It is based on the idea to separate the update and access to

dynamic predicates. Updates can only be performed on the top-level in the I/O monad in order to ensure a well-defined sequence of updates. The access to dynamic predicates is initiated also in the I/O monad in order to get a well-defined set of visible facts for dynamic predicates. However, the actual access can be done at any execution time since the visibility of facts is controlled by time stamps. This is important in the presence of an advanced operational semantics (demand-driven evaluation) where the actual sequence of evaluation steps is difficult to determine in advance.

Furthermore, dynamic predicates can be also persistent so that their definitions are externally stored and recovered when programs are restarted. We have sketched an implementation of this concept in a Prolog-based compiler which is freely available with the current release of PAKCS [14].

Although the use of our concept is quite simple (one has to learn only three basic I/O actions), it is quite powerful at the same time since the applications of logic programming to declarative knowledge management can be directly implemented with this concept. We have used this concept in practice to implement a bibliographic database system and obtained quite satisfying results. The loading of the database containing almost 10,000 bibliographic entries needs only a few milliseconds, and querying all facts is also performed in milliseconds due to the fact that they are stored in main memory.

References

1. J.M. Almendros-Jiménez and A. Becerra-Terón. A Safe Relational Calculus for Functional Logic Deductive Databases. *Electronic Notes in Theoretical Computer Science*, Vol. 86, No. 3, 2003.
2. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000.
3. S. Antoy and M. Hanus. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. International Workshop on Frontiers of Combining Systems (FroCoS'2000)*, pp. 171–185. Springer LNCS 1794, 2000.
4. S. Antoy and M. Hanus. Functional Logic Design Patterns. In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, pp. 67–87. Springer LNCS 2441, 2002.
5. B. Braßel, M. Hanus, and F. Huch. Encapsulating Non-Determinism in Functional Logic Computations. In *Proc. 13th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2004)*, pp. 74–90, Aachen (Germany), 2004. Technical Report AIB-2004-05, RWTH Aachen.
6. J. Correas, J.M. Gómez, M. Carro, D. Cabeza, and M. Hermenegildo. A Generic Persistence Model for (C)LP Systems (and Two Useful Implementations). In *Proc. of the Sixth International Symposium on Practical Aspects of Declarative Languages (PADL'04)*, pp. 104–119. Springer LNCS 3057, 2004.
7. S.K. Das. *Deductive Databases and Logic Programming*. Addison-Wesley, 1992.
8. R. Echahed and W. Serwe. Combining Mobile Processes and Declarative Programming. In *Proc. of the 1st International Conference on Computation Logic (CL 2000)*, pp. 300–314. Springer LNAI 1861, 2000.
9. H. Gallaire and J. Minker, editors. *Logic and Databases*, New York, 1978. Plenum Press.

10. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, Vol. 19&20, pp. 583–628, 1994.
11. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pp. 80–93, 1997.
12. M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pp. 47–62. Springer LNCS 1753, 2000.
13. M. Hanus. High-Level Server Side Web Scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pp. 76–92. Springer LNCS 1990, 2001.
14. M. Hanus, S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2004.
15. M. Hanus and F. Steiner. Controlling Search in Declarative Programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, pp. 374–390. Springer LNCS 1490, 1998.
16. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8). Available at `http://www.informatik.uni-kiel.de/~curry`, 2003.
17. D. Leijen and E. Meijer. Domain Specific Embedded Compilers. In *Proceedings of the 2nd Conference on Domain-Specific Languages (DSL'99)*, pp. 109–122. ACM SIGPLAN Notices 35(1), 1999.
18. T.G. Lindholm and R.A. O'Keefe. Efficient Implementation of a Defensible Semantics for Dynamic Prolog Code. In *Proc. Fourth International Conference on Logic Programming (Melbourne)*, pp. 21–39. MIT Press, 1987.
19. J. Lloyd. Programming in an Integrated Functional and Logic Language. *Journal of Functional and Logic Programming*, No. 3, pp. 1–49, 1999.
20. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pp. 244–247. Springer LNCS 1631, 1999.
21. L. Naish. Automating control for logic programs. *Journal of Logic Programming (3)*, pp. 167–183, 1985.
22. S.L. Peyton Jones and J. Hughes. Haskell 98: A Non-strict, Purely Functional Language. `http://www.haskell.org`, 1999.
23. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, Vol. 29, No. 1-3, pp. 17–64, 1996.
24. P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, Vol. 29, No. 3, pp. 240–263, 1997.

Encapsulating Non-Determinism in Functional Logic Computations*

Bernd Braßel Michael Hanus Frank Huch

Institute of Computer Science, CAU Kiel, Olshausenstr. 40, D-24098 Kiel, Germany
{bbr,mh,fhu}@informatik.uni-kiel.de

Abstract. One of the key features of the integration of functional and logic languages is the access to non-deterministic computations from the functional part of the program. In order to ensure the determinism of top-level computations in a functional logic program, which is usually a monadic sequence of I/O operations, one has to encapsulate the non-determinism (i.e., search for solutions) of logic computations. However, an appropriate approach to encapsulation can be quite subtle if some subexpressions are shared, as in lazy evaluation strategies. In this paper we propose a new approach to encapsulate non-deterministic computations for the declarative multi-paradigm language Curry. It is based on providing a primitive I/O action for encapsulation from which various specialized search operators can be derived. In order to provide a formal foundation for this new approach to encapsulation we define its operational semantic.

* This work has been partially supported by the DFG under grant Ha 2457/1-2.

W. Goerigk (Hrsg.): Programmiersprachen und Rechenkonzepte. Bericht Nr. 0410, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

Tracing Curry by Program Transformation

Frank Huch

Institut für Informatik, CAU Kiel, Olshausenstr. 40, D-24098 Kiel, Germany.
{bb,mh,fhu}@informatik.uni-kiel.de

Abstract

Zum Debuggen deklarativer, insbesondere nicht-strikter funktionaler und funktional-logischer Sprachen hat sich gezeigt, daß der Standardansatz mit schrittweiser Programmausführung und Visualisierung der aktuellen Konfiguration nicht zur Fehlerfindung geeignet ist. Insbesondere durch die verzögerte Auswertung ist die Reduktionssemantik nur schwer nachzuvollziehen. Als ein möglicher alternativer Ansatz wurde *Tracing* vorgeschlagen, bei dem ein Programmlauf aufgezeichnet wird und dieser nach dem Programmende (hierbei sind auch Laufzeitfehler und Programmunterbrechung möglich) mit speziellen Tools analysiert werden kann. So kann die verzögerte Auswertung beispielsweise wie eine strikte Auswertung durchlaufen werden, wobei komplette Unterberechnungen wie bei Standarddebuggern für imperative Sprachen "geskipt" werden können. Auch sind andere Darstellungen, wie z.B algorithmisches Debuggen oder eine Bottom-Up-Sicht der Berechnungsergebnisse möglich.

Dieser Vortrag beschäftigt sich mit der Erzeugung solcher Traces für die nicht-strikte funktional-logische Sprache Curry. Neben einem interpreterbasierten Prototypen haben wir eine Programmtransformation entwickelt, mit der Curry Programme transformiert werden, so daß sie als Seiteneffekt einen Trace in eine Datei schreiben. Ähnliche Ansätze gibt es bereits im Hat-System für die Programmiersprache Haskell. Bei der Erweiterung auf Curry müssen insbesondere die logischen Features, wie nichtdeterministische Auswertung und logische Variablen, berücksichtigt werden.

W. Goerigk (Hrsg.): Programmiersprachen und Rechenkonzepte. Bericht Nr. 0410, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

Pair Evaluation Algebras in Dynamic Programming

Robert Giegerich and Peter Steffen

Faculty of Technology, Bielefeld University
Postfach 10 01 31, 33501 Bielefeld, Germany
{robert,psteffen}@techfak.uni-bielefeld.de

Abstract. Dynamic programming solves combinatorial optimization problems by recursive decomposition and tabulation of intermediate results. The recently developed discipline of algebraic dynamic programming (ADP) helps to make program development and implementation in nontrivial applications much more effective. It raises dynamic programming to a declarative level of abstraction, separates the search space definition from its evaluation, and thus yields more reliable and versatile algorithms than the traditional dynamic programming recurrences. Here we extend this discipline by a pairing operation on evaluation algebras, whose clue lies with an asymmetric combination of two different optimization objectives. This leads to a surprising variety of applications without additional programming effort.

1 Introduction

1.1 Motivation

Dynamic Programming is an elementary and widely used programming technique. Introductory textbooks on algorithms usually contain a section devoted to dynamic programming, where simple problems like matrix chain multiplication, polygon triangulation or string comparison are commonly used for the exposition. This programming technique is mainly taught by example. Once designed, all dynamic programming algorithms look kind of similar: They are cast in terms of recurrences between table entries that store solutions to intermediate problems, from which the overall solution is constructed via a more or less sophisticated case analysis. However, the simplicity of these small programming examples is deceiving, as this style of programming provides no abstraction mechanisms, and hence it does not scale up well to more sophisticated problems.

In biological sequence analysis, for example, dynamic programming algorithms are used on a great variety of problems, such as protein homology search, gene structure prediction, motif search, analysis of repetitive genomic elements, RNA secondary structure prediction, or interpretation of data from mass spectrometry [6, 2]. The higher sophistication of these problems is reflected in a large number of recurrences – sometimes filling several pages – using more complicated case distinctions, numerous tables and elaborate scoring schemes.

W. Goerigk (Hrsg.): Programmiersprachen und Rechenkonzepte. Bericht Nr. 0410, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001

An *algebraic* style of *dynamic programming* (ADP) has recently been introduced, which allows to formulate dynamic programming algorithms over sequence data on a more convenient level of abstraction [4, 5]. In the ADP approach, the issue of scoring is cast in the form of an *evaluation algebra*, the logical problem decomposition is expressed as a *yield grammar*. Together they constitute a declarative, and notably subscript-free problem specification that transparently reflects the design considerations. Written in a suitable notation, these specifications can be implemented automatically – often more efficiently and always more reliably than hand-programmed DP recurrences.

In this paper, we extend the ADP discipline by one further operator, a product construction on evaluation algebras. Its clue lies with an asymmetric, nested definition of the product of two objective functions, which allows to optimize according to a primary and a secondary objective. Beyond this, when using some non-optimizing algebras, it leads to an unexpected variety of applications, such as backtracing, multiplicity of answers, ambiguity checking, and more.

1.2 Basic terminology

Alphabets. An *alphabet* $\mathcal{A}$ is a finite set of symbols. Sequences of symbols are called strings. ε denotes the empty string, $\mathcal{A}^1 = \mathcal{A}$, $\mathcal{A}^{n+1} = \{ax | a \in \mathcal{A}, x \in \mathcal{A}^n\}$, $\mathcal{A}^+ = \bigcup_{n \geq 1} \mathcal{A}^n$, $\mathcal{A}^* = \mathcal{A}^+ \cup \{\varepsilon\}$.

Signatures and algebras. A *signature* Σ over some alphabet $\mathcal{A}$ consists of a sort symbol S together with a family of operators. Each operator o has a fixed arity $o : s_1 ... s_{k_o} \to S$, where each s_i is either S or $\mathcal{A}$. A *Σ-algebra* $\mathcal{I}$ over $\mathcal{A}$, also called an interpretation, is a set $\mathcal{S}_\mathcal{I}$ of values together with a function $o_\mathcal{I}$ for each operator o. Each $o_\mathcal{I}$ has type $o_\mathcal{I} : (s_1)_\mathcal{I} ... (s_{k_o})_\mathcal{I} \to \mathcal{S}_\mathcal{I}$ where $\mathcal{A}_\mathcal{I} = \mathcal{A}$.

A *term algebra* T_Σ arises by interpreting the operators in Σ as *constructors*, building bigger terms from smaller ones. When variables from a set V can take the place of arguments to constructors, we speak of a term algebra with variables, $T_\Sigma(V)$, with $V \subset T_\Sigma(V)$. By convention, operator names are capitalized in the term algebra.

Trees and tree patterns. Terms will be viewed as rooted, ordered, node-labeled trees in the obvious way. All inner nodes carry (non-nullary) operators from Σ, while leaf nodes carry nullary operators from Σ or symbols from $\mathcal{A}$. A term/tree with variables is called a *tree pattern*. A tree containing a designated occurrence of a subtree t is denoted $C[...t...]$.

A *tree language* over Σ is a subset of T_Σ. Tree languages are described by tree grammars, which can be defined in analogy to the Chomsky hierarchy of string grammars. Here we use regular tree grammars, originally studied in [1], with some algebraic flavoring added such that they describe term languages over some signature Σ and some alphabet $\mathcal{A}$.

2 Algebraic Dynamic Programming in a nutshell

ADP is a domain specific language for dynamic programming over sequence data. In ADP, a dynamic programming algorithm is specified by a yield grammar

and an evaluation algebra. The grammar defines the search space as a term language, the algebra the scoring of solution candidates. Their interface is a common signature.

Our introduction here must be very condensed. For a complete presentation, including the programming methodology that comes with ADP, the reader is referred to [5] and to the ADP website at

`http://bibiserv.techfak.uni-bielefeld.de/adp/`.

Definition 1. *(Tree grammar over Σ and $\mathcal{A}$)*
A regular tree grammar $\mathcal{G} = (V, Z, P)$ over Σ and $\mathcal{A}$ is given by

- *a set V of nonterminal symbols,*
- *a designated nonterminal symbol Z, called the axiom, and*
- *a set P of productions of the form $v \to t$, where $v \in V$ and $t \in T_\Sigma(V)$. $v \to t_1 | \ldots | t_n$ shall denote the short form for $v \to t_1, \ldots, v \to t_n$.*

The derivation relation for tree grammars is $\Rightarrow^*$, with $C[...v...] \Rightarrow C[...t...]$ if $v \to t \in P$. The language of $v \in V$ is $\mathcal{L}(v) = \{t \in T_\Sigma | v \Rightarrow^* t\}$, the language of $\mathcal{G}$ is $\mathcal{L}(\mathcal{G}) = \mathcal{L}(Z)$.

For convenience, we add a lexical level to the grammar concept, allowing strings from $\mathcal{A}^*$ in place of single symbols. $L = \{char, string, empty\}$ is the set of lexical symbols. By convention, $\mathcal{L}(char) = \mathcal{A}$, $\mathcal{L}(string) = \mathcal{A}^*$, and $\mathcal{L}(empty) = \{\varepsilon\}$.

The yield function y on the trees in T_Σ is defined by $y(a) = a$ for $a \in \mathcal{A}$, and $y(f(x_1, \ldots, x_n)) = y(x_1) \ldots y(x_n)$, for $f \in \Sigma$ and $n \geq 0$. Note that nullary constructors by definition have yield ε, hence $y(t)$ is the string of leaf symbols from $\mathcal{A}$ in left to right order.

We shall also allow conditional productions, where a simple predicate must be satisfied by the yield string derived.

Definition 2. *(Yield grammars and yield languages) Let $\mathcal{G}$ be a tree grammar over Σ and $\mathcal{A}$, and y the yield function. The pair $(\mathcal{G}, y)$ is called a yield grammar. It defines the yield language $\mathcal{L}(\mathcal{G}, y) = \{y(t) | t \in \mathcal{L}(\mathcal{G})\}$.*

Definition 3. *(Yield parsing) Given a yield grammar $(\mathcal{G}, y)$ over $\mathcal{A}$ and a sequence $w \in \mathcal{A}^*$, the yield parsing problem is to find $P_\mathcal{G}(w) = \{t \in \mathcal{L}(\mathcal{G}) | y(t) = w\}$.*

Note that the input string w is "parsed" into trees $t \in \mathcal{L}(\mathcal{G})$, each of which in turn has a tree parse according to the tree grammar $\mathcal{G}$. These tree parses must exist – they ensure that each candidate t corresponds to a proper problem decomposition – but otherwise, they are irrelevant and will play no part in the sequel. The candidate trees t, however, represent the search space spanned by a particular problem instance, and will be subject to scoring and choice under our optimization objective.

Definition 4. *(Evaluation algebra) Let Σ be a signature with sort symbol Ans. A Σ-evaluation algebra $\mathcal{I}$ is a Σ-algebra augmented with an objective function $h_\mathcal{I} : \mathcal{L}(Ans_\mathcal{I}) \to \mathcal{L}(Ans_\mathcal{I})$, where $\mathcal{L}(Ans_\mathcal{I})$ denotes the set of lists with elements from $Ans_\mathcal{I}$.*

Given that yield parsing constructs the search space for a given input, all that is left to do is to evaluate the candidates in a given algebra, and make our choice via the objective function $h_{\mathcal{I}}$.

Definition 5. *(Algebraic dynamic programming)*

- *An ADP problem is specified by a signature Σ over $\mathcal{A}$, a yield grammar $(\mathcal{G}, y)$ over Σ, and a Σ-evaluation algebra $\mathcal{I}$ with objective function $h_{\mathcal{I}}$.*
- *An ADP problem instance is posed by a string $w \in \mathcal{A}^*$. The search space it spawns is the set of all its parses, $P_{\mathcal{G}}(w)$.*
- *Solving an ADP problem is computing $h_{\mathcal{I}}\{t_{\mathcal{I}} \mid t \in P_{\mathcal{G}}(w)\}$ in polynomial time and space with respect to $|w|$.*

Bellman's Principle[1], when satisfied, allows the following implementation of tree parsing: As the trees that constitute the search space are constructed in a bottom up fashion, rather than building them explicitly as terms in T_Σ, for each constructor C the evaluation function $C_{\mathcal{I}}$ is called. Thus, the tree parser computes not trees, but answer values. To reduce their number (and thus to avoid exponential explosion) the objective function may be applied at an intermediate step where a list of alternative answers has been computed. Due to Bellman's Principle, the recursive intermediate applications of the choice function do not affect the final result.

In this paper, the reader is asked to take it for granted that the tree parsing sketched here can be implemented efficiently. ADP comes with an ASCII notation for yield grammars and evaluation algebras, which is either embedded in Haskell or directly translated to C. In the examples at the aforementioned website we explicitly annotate productions to the results of which the choice function is to be applied, but for our presentation here the reader may assume that it is applied wherever appropriate.

3 RNA secondary structure prediction

We need an example with a certain sophistication to illustrate well the variety of applications we have in mind. The following is a simplified version of the RNA structure analysis problem that plays an important role in biosequence analysis.

All genetic information in living organisms is encoded in long chain molecules. DNA is the storage form of genetic information, its shape being the double helix discovered by Watson and Crick. Mathematically, the human genome is a string of length 3×10^9 over a four letter alphabet. RNA is the active form of genetic information. It is transcribed from a segment of the DNA as a chain of *bases* or *nucleotides* A, C, G and U, denoting Adenine, Cytosine, Guanine, and Uracil. Some bases can form base pairs by hydrogen bonds: G–C, A–U and also G–U. RNA is typically single stranded, and by folding back onto itself, it forms the structure essential for its biological function. Structure formation is driven by

[1] See [5] for the formulation of Bellman's Principle in the ADP framework.

the forces of hydrogen bonding between base pairs, and energetically favorable stacking of base pairs in a helical pattern similar to DNA. While today the prediction of RNA 3D structure is inaccessible to computational methods, its *secondary structure*, given by the set of paired bases can be predicted quite reliably. Mathematically, RNA secondary structures are approximate palindromes that can be nested recursively.

In RNA structure prediction, our input sequence is a string over $\{A, C, G, U\}$. The lexical symbols *char* and *string* are renamed to *base* and *region*. The predicate *basepairing* checks whether the two bases mentioned in a production can actually form a base pair.

The first approach to RNA structure prediction was based on the idea of maximizing the number of base pairs [8]. Figure 1 (top) shows the grammar `nussinov78` which implements the algorithm of [8], with the evaluation algebra designed for maximizing the number of base pairs.

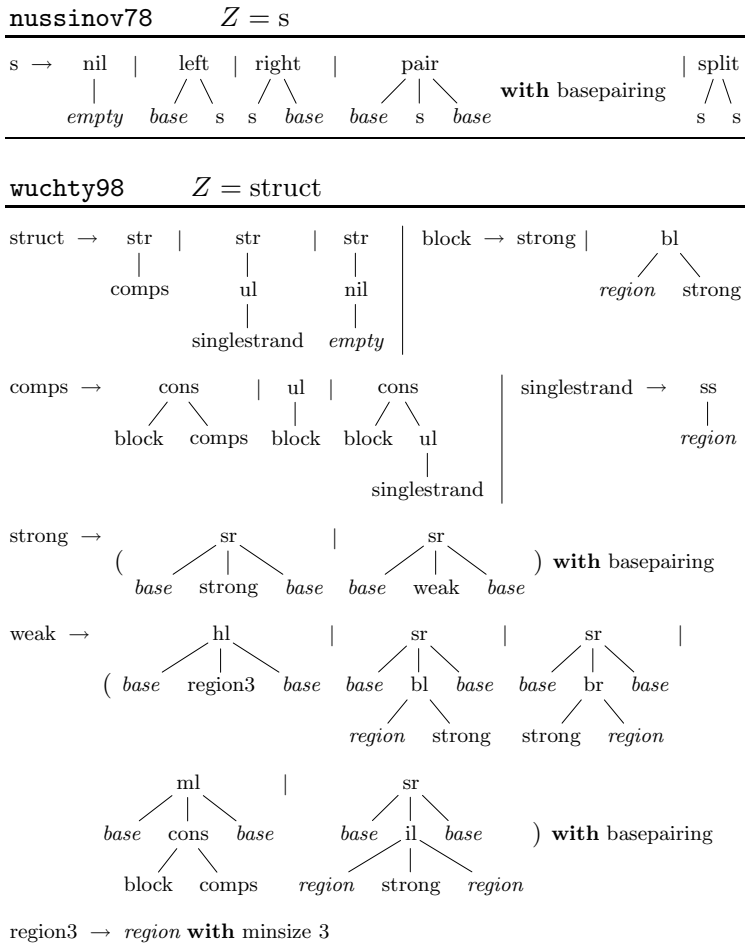

Fig. 1. Yield grammars `nussinov78` and `wuchty98`. Terminal symbols in italics.

Note that the case analysis in the Nussinov algorithm is redundant – even the base string "A" is assigned the two structures `Left('A', Nil)` and `Right(Nil, 'A')`, which actually denote the same shape.

Base pair maximization ignores the favorable energy contributions from base pair stacking, as well as the unfavorable contributions from loops. A non-redundant algorithm based on energy minimization was presented by Wuchty et al. [9]. Figure 1 (bottom) shows the grammar `wuchty98`. Here the signature has 8 operators, each one modeling a particular structure element, plus the list constructors (`nil, ul, cons`) to collect sequences of components in a unique way. This grammar, because of its non-redundancy, can also be used to study combinatorics, such as the expected number of feasible structures of a particular sequence of length n.

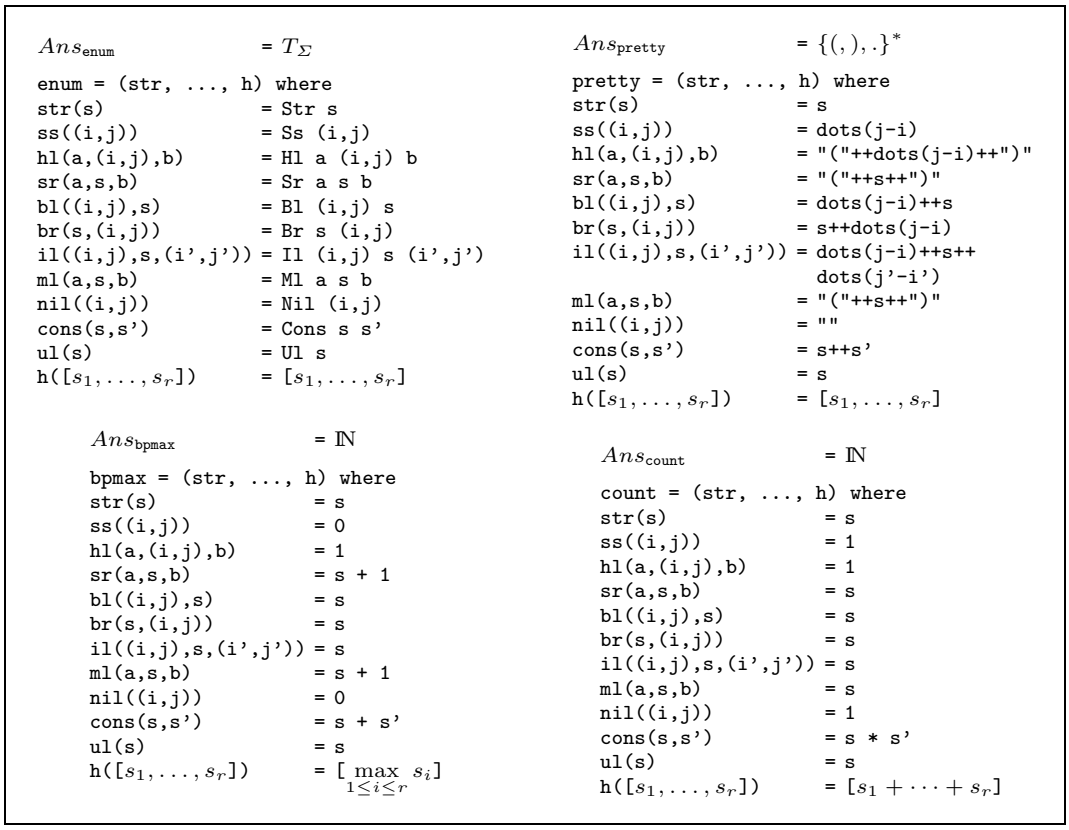

Fig. 2. Four evaluation algebras for grammar `wuchty98`. Arguments `a` and `b` denote bases, `(i,j)` represents the input subword $x_{i+1} \ldots x_j$, and `s` denotes answer values. Function `dots(r)` in algebra `pretty` yields a string of `r` dots (`'.'`).

This algorithm is widely used for structure prediction via energy minimization. Unfortunately, the thermodynamic model is too elaborate to be presented here, and we will stick with base pair maximization as our optimization objective for the sake of this presentation. Figure 2 shows four evaluation algebras that we will use with grammar `wuchty98`. We illustrate their use via the following examples, where `g(a,x)` denotes the application of grammar `g` and algebra `a` to input `x`, as defined in Definition 5. Appendix A shows all results for an example sequence.

`wuchty98(enum,x)`: the enumeration algebra `enum` yields unevaluated terms. Since the choice function is identity, this call enumerates all candidates in the

search space spanned by `x`. This is mainly used in program debugging, as it visualizes the search space actually traversed by our program.

`wuchty98(pretty,x)`: the pretty-printing algebra `pretty` yields a string representation of the same structures as the above, but in the widely used notation `"..(((...)).)"`, where pairing bases are indicated by matching brackets.

`wuchty98(bpmax,x)`: the base pair maximization algebra is `bpmax`, such that this call yields the maximal number of base pairs that a structure for `x` can attain. Here the choice function is maximization, and it can be easily shown to satisfy Bellman's Principle. Similarly for grammar `nussinov78`.

`wuchty98(count,x)`: the counting algebra is `count`. Its choice function is summation, and $t_{count} = 1$ for all candidates t. However, the evaluation functions are written in such a way that they satisfy Bellman's Principle. Thus, `[length(wuchty98(enum,x))] == wuchty98(count,x)`, where the righthand side is polynomial to compute, while the lefthand side typically is exponential due to the large number of answers.

4 Pair evaluation algebras

We now create an algebra of evaluation algebras by introducing a product operation `***` that joins two evaluation algebras into a single one.

Definition 6. *(Product operation on evaluation algebras) Let M and N be evaluation algebras over Σ. Their product M***N is an evaluation algebra over Σ and has the functions $f_{M,N}((m_1,n_1)...(m_k,n_k)) = (f_M(m_1,...,m_k), f_N(n_1,...,n_k))$ for each f in Σ, and the choice function $h_{M,N}([(m_1,n_1)...(m_k,n_k)]) = [(l,r)|l \in L, r \in h_N([r|(l,r) \leftarrow [(m_1,n_1)...(m_k,n_k)], l \in L])]$ where $L = h_M([m_1,...,m_k])$.*

Above, $\in$ denotes set membership and hence ignores duplicates[2], while $\leftarrow$ denotes list membership and respects duplicates. Our first observation is that this definition preserves identity and ordering:

Theorem 1. *(1) For any algebras M and N, and answer list x, $(id_M$***$id_N)(x)$ is a permutation of x. (2) If h_M and h_N minimize wrt. some order relations $\leq_M$ and $\leq_N$, then $h_{M,N}$ minimizes wrt. the lexicographic ordering $(\leq_M, \leq_N)$. (3) If both M and N minimize and satisfy Bellman's Principle, then so does M***N.*

Proof. (1) According to Def. 6, the elements of x are merely re-grouped according to their first component. For this to hold, it is essential that duplicate entries in the first component are ignored. (2) follows directly from Def. 6. (3) In the case of minimization, Bellman's Principle is equivalent to (strict) monotonicity of f_M and f_N with respect to $\leq_M$ and $\leq_N$, and this carries over to the combined functions (trivially) and the lexicographic ordering (because of (2)). □

In the above proof, *strict* monotonicity is required only if we ask for multiple optimal, or the k best, solutions rather than a single optimal one [7].

[2] This may require some extra effort in the implementation, but when a choice function does not produce duplicates anyway, it comes for free.

Theorem 1 essentially says that `***` behaves as expected in the case of optimizing evaluation algebras. This is very useful, but not too surprising. The interesting situations are when `***` is used with algebras that do not do optimization, like `enum`, `count`, and `pretty`. Applications of pair algebras are subject to the following

Proof scheme: *The declarative semantics of (i.e. the problem specified by)* $\mathcal{G}(M***N, x)$ *is given by Definition 5. Its operational semantics (the tabulating yield parser) is correct only if* $M***N$ *satisfies Bellman's Principle. This requires an individual proof unless covered by Theorem 1.*

That a proof is required in general is witnessed by the fact that, for example, `wuchty98(count***count,x)` delivers no meaningful result.

With this in mind, we now turn to applications of pair algebras. Appendix A shows all results for an example RNA sequence.

Application 1: Backtracing and co-optimal solutions Often, we want not only the optimal answer value, but also a candidate which achieves the optimum. We may ask if there are several such candidates. If yes, we may want to see them all, maybe even including some near-optimal candidates. They can be retrieved if we store a table of intermediate answers and backtrace through the optimizing decisions made. This backtracing can become quite intricate to program if we ask for more than one candidate. There are simpler ways to answer these questions:

`wuchty98(bpmax***count,x)` computes the optimal number of base pairs, together with the number of candidate structures which achieve it.

`wuchty98(bpmax***enum,x)` computes the optimal number of base pairs, together with all structures for `x` that achieve this maximum, in their representation as terms from T_Σ.

`wuchty98(bpmax***pretty,x)` does the same as the previous call, producing the string representation of structures.

It is a nontrivial consequence of Definition 6 that the above pair algebras in fact give multiple co-optimal solutions. Should only a single one be desired, we would use `enum` or `pretty` with a choice function h that retains only one (arbitrary) element. Note that our replacement of backtracing by a "forward" computation does not affect asymptotic efficiency.

Application 2: Testing ambiguity Dynamic programming algorithms can often be written in a simpler way if we do not care whether the same solution is considered many times during the optimization. This does not affect the overall optimum. A dynamic programming algorithm is then called redundant or ambiguous. In such a case, the computation of a list of near-optimal solutions is cumbersome, as it contains duplicates whose number often has an exponential growth pattern. Also, search space statistics become problematic – for example, the counting algebra speaks about the algorithm rather than the problem space, as it counts considered, but not necessarily distinct solutions. Yield grammars with a suitable probabilistic evaluation algebra implement stochastic context free grammars. The frequently used statistical scoring schemes, when trying to find the answer of maximal probability (the Viterbi algorithm, cf. [2]), are fooled by the presence

of redundant solutions. In principle, it is clear how to control ambiguity [3]. One needs to show unambiguity of the *tree* grammar[3] in the language theoretic sense, and the existence of an injective mapping from T_Σ to a canonical model of the search space. However, the proofs involved are not trivial. On the practical side, one would like to implement a check for ambiguity in the implementation of the ADP approach, but this is rendered futile by the following observation:

Theorem 2. *Non-redundancy in dynamic programming is formally undecidable.*

Proof. For lack of space, we can only sketch the idea of the proof. Ambiguity of context free language is a well-known undecidable problem. For an arbitrary context free grammar, we may construct an ADP problem where the context free language serves as the canonical model, and show that the language is unambiguous if and only if the ADP problem is non-redundant. □

Given this situation, we turn to the possibility of testing for (non-)redundancy. The required homomorphism from the search space to the canonical model may be coded as another evaluation algebra. This is, for example, the case with `pretty`. A pragmatic approach to this question of ambiguity is then to test

`wuchty98(pretty***count,x)` on a number of inputs `x`. If any count larger than 1 shows up in the results, we have found a case of ambiguity. Clearly, this test can be automated.

Application 3: Classification of candidates A `shape` algebra is a version of `pretty` that maps structures onto more abstract shapes. This allows to analyze the number of possible shapes, the size of their membership, and the (near-)optimality of members. Let `bpmax(k)` be `bpmax` with a choice function that retains the best `k` answers (without duplicates).

`wuchty98(shape***count,x)` computes all the shapes in the search space spanned by `x`, and the number of structures that map onto each shape.

`wuchty98(bpmax(k)***shape,x)` computes the best `k` base pair scores, together with their candidate's shapes.

`wuchty98(bpmax(k)***(shape***count),x)` computes base pairs and shapes as above, plus the number of structures that achieve this number of base pairs in the given shape.

`wuchty98(shape***bpmax,x)` computes for each shape the maximum number of base pairs among all structures of this shape.

5 Conclusion

We hope to have demonstrated that the evaluation algebra product as introduced here adds a significant amount of flexibility to dynamic programming. The mathematical properties of `***` are not yet fully explored. Moreover, Definition 6 is not without alternatives. One might consider to make in M`***`N the

[3] Not the *yield* grammar – it is always ambiguous, else we did not have an optimization problem.

results of M available to the choice function h_N. This leads to parameterized evaluation algebras and is a challenging subject for further study.

A Results for examples

The following table shows the application of grammar `wuchty98` with different algebras on input `x = cgggauaccacu`.

Algebra	Result
`enum`	`[Str (Ul (Bl (0,1) (Sr 'g' (Hl 'g' (3,10) 'c') 'u'))),Str (Ul (Bl (0,2) (Sr ....]`
`pretty`	`[".((.......))","..((......))",".((....))...","..((...))...",".........."]`
`bpmax`	`[2]`
`count`	`[5]`
`count***count`	`[(1,1)]`
`bpmax***count`	`[(2,4)]`
`bpmax***enum`	`[(2,Str (Ul (Bl (0,1) (Sr 'g' (Hl 'g' (3,10) 'c') 'u')))),(2,Str (Ul (Bl (0,2) ....]`
`bpmax***pretty`	`[(2,".((.......))"),(2,"..((......))"),(2,".((....))..."),(2,"..((...))...")]`
`pretty***count`	`[(".((.......))",1),("..((......))",1),(".((....))...",1),("..((...))...",1),("..........",1)]`
`shape***count`	`[("_[_]",2),("_[_]_",2),("_",1)]`
`bpmax(5)***shape`	`[(2,"_[_]"),(2,"_[_]_"),(0,"_")]`
`bpmax(5)***(shape***count)`	`[(2,("_[_]",2)),(2,("_[_]_",2)),(0,("_",1))]`
`shape***bpmax`	`[("_[_]",2),("_[_]_",2),("_",0)]`

References

1. W. S. Brainerd. Tree generating regular systems. *Information and Control*, 14:217–231, 1969.
2. R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge University Press, 1998.
3. R. Giegerich. Explaining and controlling ambiguity in dynamic programming. In *Proc. Combinatorial Pattern Matching*, pages 46–59. Springer LNCS 1848, 2000.
4. R. Giegerich. A systematic approach to dynamic programming in bioinformatics. *Bioinformatics*, 16:665–677, 2000.
5. R. Giegerich, C. Meyer, and P. Steffen. A discipline of dynamic programming over sequence data. *Science of Computer Programming*, 51(3):215–263, 2004.
6. D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.
7. T. L. Morin. Monotonicity and the principle of optimality. *Journal of Mathematical Analysis and Applications*, 86:665–674, 1982.
8. R. Nussinov, G. Pieczenik, J.R. Griggs, and D.J. Kleitman. Algorithms for loop matchings. *SIAM J. Appl. Math.*, 35:68–82, 1978.
9. S. Wuchty, W. Fontana, I. L. Hofacker, and P. Schuster. Complete suboptimal folding of RNA and the stability of secondary structures. *Biopolymers*, 49:145–165, 1999.

An Implementation of Session Types

Matthias Neubauer

Universität Freiburg

Abstract

A session type is an abstraction of a set of sequences of heterogeneous values sent and received over a communication channel. Session types can be used for specifying stream-based Internet protocols.

Typically, session types are attached to communication-based program calculi, which renders them theoretical tools which are not readily usable in practice. To transfer session types into practice, we propose an embedding of a core calculus with session types into the functional programming language Haskell. The embedding preserves typing. A case study (a client for SMTP, the Simple Mail Transfer Protocol) demonstrates the feasibility of our approach.

This is joint work with Peter Thiemann.

W. Goerigk (Hrsg.): Programmiersprachen und Rechenkonzepte. Bericht Nr. 0410, Institut für Informatik u. Prakt. Mathematik, CAU Kiel, 2001